National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

Modular Smalltalk: A First Implementation

BY

Wade Holst

©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Master of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1993

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-88378-2

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Wade Holst

TITLE OF THESIS: Modular Smalltalk: A First Implementation

DEGREE: M.Sc. in Computing Science

YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Wade Holst

Wade Holst
Box 131
Hays, AB
T0K 1B0
(403) 725-3931

October 8, 1993

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled *Modular Smalltalk: A First Implementation* submitted by Wade Holst in partial fulfillment of the degree of M.Sc. in Computing Science.

Duane Szafron

Jonathan Schaeffer

Sun Xiaoling
Xiaoling Sun

Date: Oct. 6/93

ABSTRACT

A first implementation of the Modular Smalltalk object-oriented programming language is presented. The implementation includes an object-oriented parser, object-oriented representation for code fragments and an object-oriented C-code generator, all implemented in Smalltalk-80. This implementation validates two of the five design principles of the Modular Smalltalk language and provides a vehicle for validating the other three design principles. The macro-based C-code generator is easily adaptable to generating production code in other languages like assembler. In addition, the generation technique applies to source languages other than Modular Smalltalk. The implementation includes an efficient method dispatch based on new extensions to incremental cache table coloring.

ACKNOWLEDGEMENTS

We are indebted to Brent Knight for his preliminary work on the internal representation of MS and to Brian Wilkerson and Daniel Lanovaz for the many comments they made throughout the project.

Table of Contents

1. **Introduction** 1
2. **Modular Smalltalk** 4
2.1. Design Goals 4
2.2. An Example Module 5
2.3. Objects, Classes and Instances 5
2.4. Messages and Selectors 6
2.5. Methods and Behaviors 6
2.6. State, Variables and Scope 7
2.7. Expressions 8
2.7.1. Character groupings 8
2.7.2. Literals 8
2.7.3. White Space 9
2.7.4. Message-sends 10
2.7.5. Identifiers 10
2.8. Methods 11
2.8.1. Visibility 11
2.8.2. Method Types 11
2.9. Classes 12
2.10. Modules 13
3. **The Programming Environment** **15**
3.1. Design Approaches to Implementation 15
3.1.1. Meta-Object Design Approach 15
3.1.2. Dedicated-Structure Design Approach. 17
3.1.3. Hybrid Design Approaches. 17
3.2. Implementation Goals 18
3.3. Language Classes 19
3.3.1. Objects — MSObject 19
3.3.2. Classes — MSClass 20
3.3.3. Contexts — MSContext 20
3.3.4. Closures — MSObject 22
3.3.5. MSReturnObject 23
3.4. Program Classes 24
3.4.1. Programming Environment — MSPEnv 24
3.4.1.1. Browsing and Defining Modules 25
3.4.1.2. Literals 25
3.4.1.3. Primitives 26
3.4.1.4. Kernel module and Kernel group 27
3.4.1.5. Class Information 28
3.4.1.6. The Cache Tables 28

3.4.1.7. Miscellaneous Environment State 28
3.4.2. Scanner 29
3.4.3. MSCacheTable 30
3.4.4. MSConflictSet 30
3.5. Internal Representation 30
3.5.1. Token Classes 33
3.5.1.1. Constant Token Classes 34
3.5.1.2. Variable Token Classes 34
3.5.2. Node Classes 35
3.5.2.1. MSName 40
3.5.2.2. MSLiteral 41
3.5.2.3. MSMessageSend 42
3.5.2.4. MSReturn 43
3.5.2.5. MSAssignment 43
3.5.2.6. MSBlock 44
3.5.2.7. MSMethod 45
3.5.2.7.1. MSStateMethod 45
3.5.2.7.2. MSPrimitive, MSAbstract, MSUndefined 46
3.5.2.7.3. MSAliasedMethod 47
3.5.2.8. MSClass 47
3.5.2.9. MSModule 47
4. Method Dispatch 50
4.1. Method Dispatch Techniques 50
4.1.1. Smalltalk's Lookup Dispatch 50
4.1.2. Cache-table Dispatch 50
4.1.3. Colored Cache-table Dispatch 51
4.1.4. Incremental Coloring Cache-table Dispatch 52
4.1.5. Incremental Coloring and Partitioning Dispatch (ICP) 52
4.2. André-Royer algorithm 53
4.2.1. Actions During Class Definition 54
4.2.1.1. Inheritance Copying 54
4.2.2. Selector Color Conflicts 55
4.2.3. Actions During Method Definition 56
4.3. The ICP Algorithm 59
4.3.1. ICP Dispatch Classes 59
4.3.1.1. MSSelector 60
4.3.1.2. MSDivision 60
4.3.1.3. MSCacheTable 61
4.3.1.4. MSConflictSet 63
4.3.2. The Incremental Coloring Algorithm (ICP) 64
4.3.3. The ICP Algorithm 66

4.3.3.1. An ICP example 69
4.3.3.2. Using Partition Types to Avoid Method Dispatch 72
4.4. Space Efficiency (Tail Removal) 75
4.5. The Method Dispatch Algorithms 75
5. C-Code Generation 78
5.1. C-Structures 79
5.1.1. MSObject 79
5.1.2. Classes as MSObjects 80
5.1.3. MSPEnv 80
5.1.4. MSModule 80
5.1.5. CacheTable 81
5.1.6. MSContext 81
5.1.7. State 83
5.2. Primitive C Functions 83
5.2.1. undefinedMethod and abstractMethod 83
5.2.2. msBasicNew 84
5.2.3. Required Class primitives 84
5.3. Library Functions 84
5.3.1. execute 84
5.3.2. makeClass 84
5.3.3. makeMSPEnv 85
5.4. Code Generation 85
5.4.1. Program Code Generation 86
5.4.2. Module Code Generation 87
5.4.3. Class Definition Code Generation 88
5.4.4. Method Code Generation 88
5.4.5. Code Generation for Variables, Self and Literals 90
5.4.6. Code Generation for Literal blocks 90
5.4.7. Code Generation for Message Sends 91
5.4.8. Code Generation for Assignments and Returns 94
5.5. Optimizations 94
5.5.1. Method Dispatch 95
5.5.2. Literal Blocks and Module Expressions. 95
6. Future Directions 96
6.1. Conclusions 96
6.2. Future Directions 96
6.2.1. Static Typing 96
6.2.2. Software Verifiability 97
6.2.3. Cache Tables 97
6.2.4. Garbage Collection 97
6.2.5. MS Library and Automatic ST-MS Converter 97

6.2.6. STACC : Automated Parsing Framework and Generator 98
6.2.7. Self-Implementation 98

List of Tables

Table 1 Example Message Sends 10

Table 2 Example Method Types 12

Table 3 Colored CacheTable : Two selectors sharing same color 51

Table 4 Definitions for the André-Royer Algorithm 54

Table 5 André-Royer Partition Types for selectors 54

Table 6 Example MSCacheTable 62

Table 7 MSConflictSet before parsing native behavior 64

Table 8 MSConflictSet after native behavior 64

Table 9 Definitions for the ICP Algorithm 65

Table 10 Partition types for divisions 65

Table 11 CacheTable after b:B 69

Table 12 CacheTable after c:D 69

Table 13 CacheTable after b:C 71

Table 14 CacheTable after a:C 72

Table 15 CacheTable after a:A 72

Table 16 An ICP generated coloring for the inheritance graph of Figure 15 ... 74

Table 17 Method dispatch information 75

Table 18 Method dispatch comparison 77

Table 19 MSObject array 79

Table 20 MSPEnv structure 80

Table 21 MSModule structure 81

Table 22 CacheTable structure 81

Table 23 CacheRow structure 81

List of Figures

Figure 1 Example MS code — Numbers Module 5
Figure 2 Explicit Return Behavior 24
Figure 3 Convential parsing/code generation 31
Figure 4 Object oriented parsing/code generation 32
Figure 5 BNF Grammar Rules for Parse Tree Token Classes 34
Figure 6 BNF Grammar Rules for Parse Tree Node Classes 35
Figure 7 High-Level Node Class Relationships 38
Figure 8 Low-Level Node Class Relationships 39
Figure 9 MS Module Code — Extended Numbers Module 40
Figure 10 André-Royer Class Algorithm 56
Figure 11 The André-Royer Algorithm 56
Figure 12 Modified André-Royer Algorithm. 59
Figure 13 The ICP algorithm 66
Figure 14 Example ICP Inheritance Graphs 71
Figure 15 An example inheritance graph with an inheritance exception 74
Figure 16 Algorithmic description of *init_mspenv* function 86
Figure 17 Example Code Generated for Module Function 87
Figure 18 Example Code Generated for Class Function 88
Figure 19 Example Code Generated for Block Method 90
Figure 20 Example Code Generated for Simple Message Send 92
Figure 21 Example Code Generated for Complex Message Send 93
Figure 22 Example Code Generated for Explicit Return 94
Figure 23 BNF Grammar Rules for Parse Tree Token Classes 100
Figure 24 BNF Grammar Rules for Parse Tree Node Classes 101
Figure 25 Method Dispatch Algorithm For MS using Two Cache Tables 104
Figure 26 Class Look-up Dispatch Algorithm for Multiple Inheritance 105

Chapter 1 : Introduction

Modular Smalltalk (MS) [Tek89] is an object-oriented descendant of the Smalltalk-80 (ST-80) [GR89] programming language. This thesis describes a first implementation of MS (there are no previously published implementations, although there may be some unpublished ones in the commercial sector).

Since the MS language attempts to correct the problems of Smalltalk without losing any of its advantages, MS provides both a programming environment for rapid prototyping and code generation for efficient stand-alone execution. To provide these capabilties, there are two different base languages in which MS code can be executed. ST80 was used to implement the programming environment, and MS programs can be interpreted by appropriate ST80 code, providing MS with rapid prototyping facilities. C is used to implement the stand-alone application capability and MS programs can be executed by having the PE generate C-code. The C-code is compiled together with all needed primitives and library functions to produce an executable file. The purpose of the PE is rapid prototyping, and thus execution efficiency is not a priority within the PE. The purpose of the C code is to provide an efficient implementation, and thus execution efficiency is the biggest priority during C-code generation.

The programming environment (PE), is responsible for parsing MS programs and providing an environment in which the program can be interpreted and debugged. The PE was written in ST80, chosen to allow rapid experimentation with different implementation strategies and interpretations of the language specification. Although other implementation languages would have resulted in faster parsing and interpretation, speed of interpretation was not the primary consideration. In addition, all of the classes designed for debugging Smalltalk programs can be used without change to debug MS programs, since MS debugging methods can be implemented as primitives which call the associated ST-80 methods.

Although efficiency was not a high priority during parsing and interpretation, it is of paramount interest when a production program is executed. Thus, the implementation provides a technique for generating macro-based C code for any MS program. A small library of essential miscellaneous C functions, along with the C-code that is generated, is compiled together with the applications's primitive C functions to create a stand-alone application.

Chapter 2 gives the specification of the MS language. Since this thesis attempts to be self-contained with respect to the language, it augments and expands on the initial language proposal found in [Tek89].

Chapter 3 describes the internal representation of MS programs as ST-80 objects — the parse tree. The internal representation was designed to separate the extensive compile-time responsibilities from the minimal runtime responsibilities (execution). Compile time responsibilities include: symbol tables, coloring dispatch tables, dispatch optimization, decompilation, and generation of an equivalent C language program. An interpreter written in ST-80 can be used to interpret this internal form. Chapter 3 also describes the parser that translates MS to the internal form. The conventional scanner-parser-code generator approach is not used. Instead, an elegant use of object-oriented facilities implemented in ST-80 is presented. During the creation of this parsing mechanism, an automated framework for generating such parse trees was developed [SH93].

Chapter 4 describes the implemented approach to method dispatch. We have chosen an incremental coloring approach instead of the much slower dynamic look-up approach of ST-80. The given incremental coloring algorithm expands on previous algorithms [AR92] and was designed with static typing in mind. By extending the MS language to provide static typing, the dispatch algorithm can eliminate the need for lookup in many message sends.

Chapter 5 describes the C-code generation mechanism that translates the internal representation of a program into an equivalent C program that can be compiled and executed with any ANSI C compiler. The code generated is macro based to allow for implementations in an arbitrary language by defining the macros appropriately.

Chapter 6 summarizes current results and describes future directions.

Appendix A contains the BNF grammar rules of the MS language. Appendix B contains the method dispatch algorithms and ICP algorithms. Appendix C contains a sample of the C-code generated from an MS program. Appendix D contains the Object and Kernel Module Source Code. Appendix E contains a discussion of implementation dependent issues.

The research contributions of this thesis are:

1. Validation of the design goal that the execution semantics of Modular Smalltalk are consistent. The only exception is for class extensions, which are not currently implemented. No inconsistencies are expected for class extensions,
2. Partial validation of the design goal that an efficient implementation of Modular Smalltalk is possible. Comparisons between the ICP dispatch approach and convential lookup dispatch have been made, showing the former to be substantially better. Since method dispatch is the most significant factor in slowing down the speed of Smalltalk, this result is important. However, general program efficiency benchmarks have not yet been obtained against C++, Eiffel and Smalltalk.
3. The generation of a platform to support the validation of the other three design goals of Modular Smalltalk: increased programmer productivity through code reuse and code redefinition, design and implementation efficiency for multiple programmer applications and simplicity for new users,
4. A concise description of an object-oriented parser, which can serve as the basis for a parser framework,
5. Corrections to the André-Royer [AR92] incremental coloring algorithm and extensions that support inheritance exceptions, and
6. A new macro approach to C-code generation that can easily be modified to generate assembly language code and could be adapted to other programming environments.

Chapter 2 : Modular Smalltalk

The initial language proposal for Modular Smalltalk is in [Tek89]. However, since this thesis attempts to be self-contained with respect to Modular Smalltalk, this chapter is a superset of the above technical report. A working knowledge of Smalltalk will be beneficial for a full understanding of this thesis.

Appendix A gives the complete BNF grammar specification, Appendix D describes the behavior currently defined in the Object and Kernel Modules, and Appendix E lists the implementation dependent issues of the language specification and the approach taken for them.

2.1. Design Goals

Several desirable goals for an object-oriented language guided the specification of Modular Smalltalk. These goals were [WBW88]:

1. to provide a simple, consistent execution semantics,
2. to increase programmer productivity through code reuse and code re-definition,
3. to provide efficient facilities for design and implementation in multiple programmer applications,
4. to provide for efficient implementations, and
5. to be simple enough for new users to learn easily.

Smalltalk does not satisfy these goals by failing to :

1. maintain a clear distinction between the language specification, its implementation and its development environment,
2. define explicit language semantics, independent of any implementation, and
3. support the development of application programs that execute independently of their development environment.

2.2. An Example Module

Below is a small MS module (MS programs consist soley of a collection of modules). This code will be used as a reference throughout the thesis.

```
{   module 'Numbers'                                        "Module Start"

    Object -> { from 'Kernel' }                             "Import Declaration"

    Complex (public) ->                                     "Class Declaration Start"
    {
        class { refines Object }                            "Superclass Declaration"

        instance                                            "Instance Behavior Start"
        { behavior
            { real real: } -> variable                      "State Method Declaration"
            { imag imag: } -> variable

            + -> method                                     "Binary Block Method Start"
            [ :aComplex |                                   "Formal Parameters"
                | result |                                  "Temporary Variable"

                result := Complex                           "Assignment"
                        real: (self real + aComplex real)   "Unary & Binary Message Send"
                        imag: (self imag + aComplex imag).  "Keyword Message Send"
                ^result                                     "Explicit Return"
            ]                                               "Binary Block Method End"
        }                                                   "Instance Behavior End"

        class                                               "Class Behavior Start"
        { behavior
            real:imag: ->                                   "Keyword Block Method Start"
            [ :real :imag |                                 "Method Arguments"
                Complex new                                 "Unary Message Send"
                    real: real; imag: imag.                 "Cascaded Message Send"
            ]                                               "Keyword Block Method End"
        }                                                   "Class Behavior End"
    }                                                       "Class Declaration End"

    i -> { expression Complex real: 0 imag: 1 }             "Module Expression"
}                                                           "Module End"
```

Figure 1 : Example MS code — Numbers Module

Each line in the example has a comment attached to it. In the discussion below, when a specific part of the example is being presented, the line will be identified by its associated comment. For example, to direct attention to the line in the figure containing the literal objects 0 and 1, the reference used would be (Figure 1: *Module Expression* — 0 and 1 are literal objects).

2.3. Objects, Classes and Instances

An *object* is an encapsulation of data together with the procedures that operate on that data. All data that can be referred to during program execution is represented by an object. An object is created dynamically by other objects during execution or is statically created via syntactic constructs. Objects exist at least as long as there is a reference to them (within some other object).

There are two different types of object in MS : instance objects and class objects. A *class object* acts as a template, specifying the behavior of its instances. A class object can have an arbitrary number of associated instance objects. An *instance object* represents one possible instantiation of its associated class object.

2.4. Messages and Selectors

A *message send* consists of a receiver object and a message. A *message* is a request to an object to perform a specific operation, and consists of a selector and a list of zero or more arguments, all of which are objects. A *selector* is a name used to identify to the receiver the operation being requested. A *method selector* is the object representing a selector, and thus **MethodSelector** is a required class. The receiver object and the method selector uniquely identify a method to execute. There are three types of method selector: *unary*, *binary*, and *keyword*. A unary selector consists solely of a name, and has zero arguments. A binary selector consists of one or more operator characters, and has 1 argument. A keyword selector consists of one or more names separated by colons, and has as many arguments as there are colons. The precedence of message sends is the same as in ST-80 — from highest to lowest : unary, binary, keyword. Selector names follow the same rules as identifiers. Section 2.7.4 contains some examples.

2.5. Methods and Behaviors

A method is the executable representation of an operation, and can retrieve state information from the receiver object, modify the state of the receiver object, and initiate other messages. A method is defined by a *method declaration*, which associates a selector with a method (Figure 1: *State Method Declaration, Binary Block Method Start, Keyword Block*

Method Start). A method consists of zero or more parameters (Figure 1: *Method Arguments*) and an executable body. Each method returns a single object. A message is said to *return* this object when method execution is complete. The object to be returned can be specified explicitly via the '^' token (Figure 1: *Explicit Return*) but is implicitly defined as the result of the last expression executed within the method (Figure 1: *Keyword Block Method End*) for methods without explicit returns. Note that the implicit return value is different than most object-oriented languages, for whom the default return object is the receiver. If the operation has no expressions, the destinguished value *nil* is returned. Methods can be recursive.

2.6. State, Variables and Scope

Unlike most other O-O languages, MS has no instance variables. Instead, state is represented by groups of messages that are implemented as stored methods as opposed to computed methods. *Named instance variables* from Smalltalk (or *member-data* from C++) are replaced by a pair of messages which provide access and change behavior (Figure 1: *State Method Declarations*). Thus, state method declarations explicitly define the message selectors needed to access and change specific state, implicitly increase the amount of space used by objects and implicitly define the behavior associated with the access and change method selectors specified. The *change state method* (Figure 1: *State Method Declaration — real:* and *imag:*) has as its single argument an object that will be used as the new state value of the receiving object and the return value of this method is this new state value. The return value of the *access state method* (Figure 1: *State Method Declaration — real* and *imag*) is the last value assigned by the corresponding assignment message, or nil if the assignment message has not yet been sent (Figure 1: *Unary & Binary Message Sends* — the unary messages are state method message sends).

Indexed instance variables from Smalltalk are replaced by a set of four messages that define an indexed state. An accessing method uses a single-keyword selector and its argument specifies which part of the state to access. Two different types of indexed state exist: object-valued and byte-valued. If no value has been set at the given index, the value nil is returned for object-valued states, and the value 0 is returned for byte-valued states. Object-valued states may assume any value, whereas byte-valued states are restricted to objects representing integers in the range 0 to 255 inclusive. The *assignment state method* uses a two-keyword selector where the first argument is an index that identifies which index of the indexed state and the second argument is the new value for that index of the state. The method returns the second argument. Two other selectors are associated with indexed states. The *size accessing state method* returns the number of indexable states associated with the two selectors above. The *size changing state method* sets the total number of indexable states dynamically. A typical example of such variables exist in the Array class (Appendix D), which defines an indexed instance variable. The index modifying message selector is *at:*, the index changing message is *at:put:*, the size accessing message is *size*, and the size changing method is *size:*.

2.7. Expressions

An expression is a construct denoting a rule of computation for obtaining a value by sending zero or more messages. Expressions are used in method declarations and expression declarations to perform computations.

2.7.1. Character groupings

To facilitate the presentation, various hierarchial groupings for the ASCII characters are given names. These groupings are used to describe tokens. For a full description refer to Appendix A.

2.7.2. Literals

MS provides for literal representations of characters, integers, floating point numbers, strings, selectors, and arrays of these literals. Such literals are immutable, so the object represented by a literal is always apparent from its lexical representation. The result of attempting to modify literals is implementation dependent. Any object with a literal representation can be used in literal arrays.

Numeric literals consists of integers and floats. The initial language specification does not specify the range of numeric values that an implementation must support. This implementation supports the same numeric ranges as does the C language. Integer literals are instances of the required class Integer. Although usually expressed in decimal (Figure 1: *Module Expression*) , they can be represented in any base between 1 and 36 by giving a radix specification after the number. The radix for bases 11 through 36 is A through Z respectively. Only digits less than the radix can be used in the integer. Floating point literals are instance of the required class Float and consists of a decimal part, fractional part and optional exponent.

Character literals are instances of the required class Character. They represent only the subset of characters in the ASCII character set which have a printable representation. Other characters may be accessed by sending a message to the class Character with an ASCII integer value as an argument. This implies that the class Character also has a required method to perform this operation. The language specification does not specify what this method should be called. Although there is no real need to specify the name, as long as the associated behavior is defined, it is advantageous that all implementations have the same names for the same required behaviors. Thus, an extension to the language specification consists of requiring that the Character class implement the method *value:*, which accepts an MS integer and returns the character associated with the ASCII code of that integer.

String literals represent instances of the required class String, and selector literals represent instances of the required class MethodSelector. Array literals represent instances of the required class Array. They can have any number of elements but all elements must be literals (including imbedded arrays).

2.7.3. White Space

MS expressions are free-format, and thus any number of separator characters or comments may appear between any two nonterminals. To save space and improve readability, the

BNF grammar in Appendix A does not show these separators.

2.7.4. Message-sends

Message sends come in three varieties, but all three have the same basic form, a receiver object followed by the message selector and arguments. Unary messages do not have any arguments. Binary messages have special message selectors and one argument. Keyword messages have an arbitrary number of keywords in their message selector, and one argument for each keyword. The following are examples of message sends :

Message Send	Description
data *size*	unary message send asking an object for its size
1 + 7	binary message send adding two literal integers
self *at:* 3 *put:* data	keyword message send modifying state
data *from:* 3 *to:* 7 *add:* 10	keyword message send

Table 1 : Example Message Sends

2.7.5. Identifiers

There are three types of identifiers: module constants (Figure 1: *Import Declaration — Object* is a module constant), method parameters (Figure 1: *Method Arguments — real* and *imag* are parameters), and temporary variables (Figure 1: *Temporary Variable — result* is a temporary variable). Of these three, only temporary variables may have their values changed by assignment, and are thus the only true variables. The values of module constants and method parameters vary only on successive executions of the module or block.

The scoping rules for temporary variables are more complex than for Smalltalk. An object bound to an identifier declared within a block is inaccessible by that identifier outside the block. The same rule applies to identifiers that are declared in modules (module constants), unless the identifier is exported by the declaring module and imported by another module. Identifier scopes can be statically nested by nesting blocks, but modules cannot be nested.

2.8. Methods

A method declaration consists of a message selector, a visibility attribute and a method.

2.8.1. Visibility

A method declared as private is only understood (executed) if its sender and receiver are in the same class. Unlike C++, there is no protected visibility mode that allows senders to be objects from subclasses. This is due to the philosophical view in MS that subclasses are clients of their superclasses with no special status beyond what any other client class can claim.

2.8.2. Method Types

There are six types of method implementation, with examples of each shown in the table below. The first three are named abstract, undefined and primitive respectively. An abstract or undefined method requires no additional syntactic information. A primitive method is uniquely specified by its class and selector and does not require any other MS information, but the code associated with the primitive method selector must be written in the appropriate base language. The fourth type of method, called an aliased method, allows behavior defined in superclasses to be associated with a new selector. The fifth type of method is a state method (Figure 1: *State Method Declaration*) that specifies part of a state declaration, as described in Section 2.5. The sixth type of method is a block method (Figure 1: *Binary Block Method Start — Binary Block Method End*). A block method consists of a list of formal parameters (Figure 1: *Formal Parameters*), a list of temporary variables (Figure 1: *Temporary Variable*), and a list of expressions (Figure 1: *Assignment, Explict Return*). When a block is evaluated, statements within the block are not actually executed. Instead, the evaluation of a block returns a closure. A closure consists of the block and a context. The context maintains the values of all variables visible within the block and provides space for their modification during a particular execution of the method represented by that block. To actually execute a method, this context is executed, which results in some unique object being returned as the result of the execution.

MethodType	Example
abstract	size -> abstract
undefined	menuList -> undefined
primitive	value: -> primitive
aliased	myNew -> alias Object new
state	{ real real: } -> variable { size size: | at: at:put: } -> variable { size size: | at: at:put: } -> binary
block	blockMethod -> method [self at: 1 put: true. ^self at: 10]

Table 2 : Example Method Types

2.9. Classes

A class declaration (Figure 1: *Class Declaration Start — Class Declaration End*) consists of an optional superclass declaration (Figure 1: *Superclass Declaration*), an optional instance behavior (Figure 1: *Instance Behavior Start — Instance Behavior End*), and an optional class behavior (Figure 1: *Class Behavior Start — Class Behavior End*). Class declarations can inherit from zero or more existing class declarations, as long as the intended superclasses are visible within the module containing the class declaration and these superclasses do not inherit from the class being defined (the inheritance hierarchy must be acyclic). Unlike Smalltalk, class declarations are static, are not objects or expressions, and do not exist during the execution of a program. Thus, classes cannot be created at run-time or have their behavior modified. However, each class itself is an object and its state can change at run time in response to class messages. The instance behavior of a class consists of the locally declared instance behavior (native behavior) and instance behaviors inherited from all superclasses. The class behavior is similarly derived.

There are four rules governing the merging of native and inherited behavior.

1. All method declarations in the native behavior are included. Any inherited method with

the same selector as a native selector is excluded.

2. Otherwise, if a selector is bound to exactly one inherited method, that binding is included. This is true even if the binding is shared by more than one superclass. A method is shared only if the superclasses in question inherited the method from the same ancestor, and none of the ancestors between that ancestor and the superclass (including itself) re-defined a selector with the same name.
3. Otherwise, if all but one of the bindings from the inherited behavior is declared as abstract, the non- abstract binding is included.
4. Otherwise, there is a conflict and the program is invalid.

The fourth rule implies that if any selector is inherited from two (or more) different superclasses, that selector must be defined within the native behavior, either explicitly or via an alias. For example, if both class A and class B define native behavior for selector *alpha*, and class C inherits from both, C must define native behavior for *alpha*. However, one possible native behavior declaration is to use the same method as is used by one of the superclasses, via an alias. Thus, the native behavior "alpha -> alias A alpha" within C will resolve the conflict.

2.10. Modules

Modules manage the scope of names. A module (Figure 1: *Module Start — Module End*) consists of an optional module name and a set of constant bindings between names and objects. Each binding has one of three forms: an import declaration, a class declaration or an expression declaration. When a module imports an object by name from another module (Figure 1: *Import Declaration*), it creates a local name for that object in the importing module. Class declarations (Figure 1: *Class Declaration Start*) provide the state and behavior of programs. Module Expressions (Figure 1: *Module Expression*) provide the starting point for execution.

Each name binding in a module has an associated visibility attribute, private or public, that determines whether other modules can import the name. Modules are not objects and do not exist during the execution of a program. Modules also provide a facility for class extension, whereby behavior can be added to a class (which was presumably imported from a different module). Class extensions are not allowed to remove or redefine existing behavior; they can only add behavior.

Chapter 3 : The Programming Environment

3.1. Design Approaches to Implementation

Before beginning a detailed discussion of the MS Programming Environment (PE), some comments on design approaches and implementation goals is merited. During the designing of the PE and C-code components of the MS language implementation, two general design approaches evolved, representing profoundly different ways of viewing the data structures used to implement the language. Both approaches have their advantages and disadvantages, as will be discussed below.

The difference between the two approaches is centered around the most critical data-structure used when implementing an object-oriented language, namely the data-structure representing an object. For MS, the name of this data-structure will be *MSObject*. In the *meta-object* approach, this is the only data-structure defined in the base language (all data-structures are treated as MSObjects), whereas in the *dedicated-structure* approach, a different structure is defined in the base language for every data-structure required.

Within the MS implementation, the following data-structures needed to be implemented: object, class, cachetable, context, context stack, closure, programming environment, module, indexed instance variable. Keep these data-structures in mind while reading the discussion below describing the two design approaches.

3.1.1. Meta-Object Design Approach

Pure object-oriented languages have an interesting advantage over other types of languages during implementation. Normally, the data structures used in the implementation language to represent run-time concepts are not accessible by the implemented language. For example, there is no way to access the assembler language data blocks used to represent a context stack for the C language from within a C program. However, since the run-time environment of a pure object-oriented language is maintained by a single data structure representing an object, it is possible to allow the language to access the structures in the underlying language which implement it.

In the *meta-object* approach to implementation, all data-structures are treated as MS classes, with individual instances of those data-structures represented as MSObjects. For example, the cache tables would be stored in an MSObject structure, whose state is appropriately defined by a MS cachetable class. *The state (and possibly behavior) of data-structures used to implement the language is defined by code written in the language being implemented.* The primary advantage of this approach is that the data-structures used to implement the language are visible from programs within the language. This in turn provides for superior debugging facilities and the potential for implementing MS in MS.

Another advantage is the improvement in code readability and maintenance by using the rigorously structured OO data-structure paradigm (state is accessed in a uniform manner that is not affected by the addition or modification of state). The disadvantage of this approach is the extra level of indirection incurred, and thus a reduction in execution efficiency. In the C-code implementation, where efficiency is paramount, this disadvantage is the critical factor in determining which approach is used. However, in the PE, the *Meta-Object* approach is desirable.

This ability to define the data-structures used during the implementation of the language in such a way that programs within the language can access them is a profound one, whose potential needs to be further explored. To take the approach to its extreme, in addition to the language implementation data-structures (i.e. state) being represented as MSObjects, *all* behavior acting on these data-structures would be defined by behavior specified by MS programs. However, using functions to perform simple accessing and changing operations on data is unacceptable from an efficiency perspective, so in the discussion below, the meta-object approach refers to data-structures as MSObjects without requiring special MS behavior to be defined in order to access these data-structures. That is, the implementation may assume a knowledge of the format of a particular MSObject in order to access part of its state without having to worry about abstractions.

3.1.2. Dedicated-Structure Design Approach.

The more conventional approach to data-structures is to define a separate data-structure within the base language for each concept necessary. This approach will be referred to as the *dedicated-structure* approach. Its primary advantage is efficiency, both from a time and a space perspective. However, from a maintenance perspecitve, it is inferior to the meta-object approach, since it requires specific code to access elements of the data-structures, which in turn results in maintenance problems if implementation data-structures need to be extended or modified. Usually this disadvantage is not serious during language implementation since the data-structures needed to implement a language do not change unless the language changes. However, as will be mentioned below, flexibility of design is an important goal of this language implementation, due to the expectation of extensions to the language. Without this implementation flexibility, language extensions could require substantial code modification.

3.1.3. Hybrid Design Approaches.

Some data-structures in the MS implementation cannot be represented as MSObjects, due to self-referencing problems. For example, an indexed instance variable cannot be represented as an instance of MS class Array, since the Array class implements its state as an indexed instance variable (requiring the indexed state data-structure in order to implement the indexed state date-structure). Thus, a pure meta-object approach is not possible. However, a combination of the two approaches can be used to provide the advantages of both. The approach taken in both the PE and C-code implementations is such a *hybrid approach*. Some data-structures are represented as MSObjects, with associated MS classes defined appropriately. Others are dedicated structures to provide for efficiency.

3.2. Implementation Goals

There were three primary goals to be achieved during implementation. However, the importance placed on individual goals differs between the PE and C-code, as will be discussed below.

1. **Implementation Flexibility**

   Flexible design to provide for rapid prototyping of different methods of implementing MS and to provide for language extensions. This implies a preference for the meta-object design approach over the dedicated-structure approach. Within the PE, explicit separation of run-time and compile-time responsibilities contributes to implementation flexibility, whereas within the C-code, the reliance on macros satisfies this goal.

2. **Efficient Execution**

   To validate the MS language specification goal of efficient implementations, execution should be optimal. This implies a preference for the dedicated-structure approach over the meta-object approach.

3. **Minimal Special-case Code**

   The amount of special case code for required classes is kept to a minimum. For example, it is desirable to have required class state and behavior be defined in exactly the same way that any other module would be defined (i.e. by writing MS source code). As will be discussed below, it is impossible to avoid some special-case code, but very little is needed.

There is an important difference in emphasis between the PE and C-code. In the C-code, execution efficiency is the highest priority, whereas in the PE, flexible design is more important. However, implementation flexibility is still an important part of the C-code, and execution efficiency is attempted within the PE when it doesn't conflict with implementation flexibility.

The rest of this chapter contains a discussion of the programming environment, including its responsibilities and the implementation approaches used. Specifically, the parsing of MS programs and the generation of base language code is discussed in detail. The approach taken towards these operations is fundamentally different than the conventional parsing and code generation paradigm.

The PE is implemented in ST80 and thus all data-structures and implementation code is defined by ST80 classes. The classes for the PE can be divided into three groups : node classes, program classes, and language classes. Node classes represent syntactic constructs of the language which have a well-defined execution semantics. Program classes maintain information about the entire program. Language classes control behavior that is general to all programs and not dependent on a specific program.

3.3. Language Classes

The current implementation of the PE has taken the *dedicated-structure* approach of representing MS objects, MS contexts and MS classes as different Smalltalk classes. MS classes are represented by a subclass of MSObject, so MS classes have the state and behavior of objects, but are represented by a class different than MSObject. Since *Closure* is a required class in MS, MS closures *must* be represented as MS objects. The various auxillary data-structures, such as cache-tables and indexed state are also dedicated structures, but subsequent versions of the PE will place more emphasis on the meta-object design approach.

3.3.1. Objects — MSObject

The most important ST-80 language class is the class responsible for representing an MS object. The state of the MSObject class consists of an *msClass* instance variable (MSClass) storing a reference to a class object, and a *state* instance variable (Array) which maintains the list of named instance variables, indexed instance variables and byte indexed instance variables, as well as the associated numbers of such variables. There are various methods of implementing the MSObject, some better from an abstraction perspective, others better from an efficiency perspective. To provide for the implementation flexibility goal mentioned above, several different implementations exist, so that comparisons can be made between them as to effects on efficiency, etc. Chapter 5 describes the most efficient implementation, which is the one used within the C-code implementation.

3.3.2. Classes — MSClass

Since MS classes are objects, the MSObject class could be used to represent them. However, the MSObject as defined above only stores that information needed by instance objects. In addition to all of the state and behavior of instance objects, class objects need to store the instance behavior and class behavior, and have behavior specific to themselves. Instead of providing additional space and behavior within each MSObject which is used only by relatively few instances, a subclass of MSObject is used to describe class objects. This new class (MSClass) inherits all the state and behavior from MSObject, then defines additional state and behavior specific to class objects. Remember that instance objects do not maintain their associated behavior. Class objects in MS are responsible for maintaining two sets of behavior, one applicable to instance objects of that class, and one applicable to the class object itself. Class objects also store their name, the number of class variables and various implementation dependent flags.

The current PE implementation is somewhat flawed, in that it has merged the node class representing class descriptions (MSClassDesc) with the run-time data structure used to represent class objects (MSClass). Future versions of the implementation will not merge them, and will instead use the implementation described above. The primary reason for keeping node classes and run-time data structures separate is the potential for automatic generation of the parsing behavior of the PE[SH93]. An indepth discussion of node classes is presented later in this chapter.

3.3.3. Contexts — MSContext

Some structure within ST80 is needed to represent contexts in MS. The narrow operational definition of a context is a structure representing all variables defined at a particular scoping level. Each variable is assigned an offset into this context. The wider operational definition of a context is actually a linked list of contexts (a static context chain), for representing all variables defined at a particular scoping level and any higher scoping level in which the current scoping level is embedded. Variables defined at any given level are assigned offsets into the context associated with that scoping level. In this way, any variable visible within a particular scoping level can be identified by a level-offset pair, specifying which context and which offset within that context to use in order to find the value of the variable. In this broader definition, contexts not only maintain values of variables, but also a static link to their parent context. During the execution of a program, there will always be a *current context*, namely the context of the block (or module) currently executing. Remember that a variable is simply a position within a static context chain, specified by a level and an offset. The level of a variable is relative to the current context, and thus the level of the same variable can differ in two different contexts. The current context is at level 0, and thus the level specifies how many times to follow the static link.

The ST80 class used to represent such contexts is MSContext. Its state includes an instance of class Array, whose indices are equivalent to the offsets assigned to MS variables, and whose elements are MSObject instances storing the values of those variables. Furthermore, the state includes an MSContext instance, and refers to the parent context of the current context (i.e. the context associated with the scope of the scope associated with the current context).

Associated with contexts is the concept of the *dynamic context stack*. This stack represents the order in which blocks are called within a program, since modules, block methods and literal blocks are uniquely identified by their associated context. Such a context stack always has a module context as its bottom element. Each time a block method or literal block is called, either by the module or by another block method or literal block, the context associated with the new block is pushed onto this dynamic context stack. Such a context stack must be explicitly maintained at run-time so that after a method finishes execution, the previous context can be made the current context again. Within the PE, contexts are passed to the ST80 methods used to implement message sending so this context stack does not need to be explicitly stored, since it is automatically maintained by the ST80 context stack. However, within the C-code, such a context stack must be explicitly maintained.

3.3.4. Closures — MSObject

Within MS, modules have a read-only context, whose values can never been modified by run-time code. Block methods and literal blocks, however, may have temporary variables, in which case their contexts can change on successive calls. A *closure* is used to uniquely identify a particular execution of a block. Thus, a closure consists of a block (some form of executable code) and a context associated with that block. All closures for a particular block will have the same block reference, but each one will have a different context. This is necessary because the language specification states that methods can be recursive. Thus, a block may exist on the dynamic context stack more than once at the same time (with different static context chains). If the context used was the same in both instances, modifications during the execution of one would make modifications in the context of the other, which is not the expected execution semantics. Furthermore, although literal blocks cannot be called directly, the block method in which they are embedded can be, so literal block contexts can also be on the dynamic stack more than once, and thus must also be unique on each occurence. There are times when unique contexts are not needed, such as when the block method or literal block does not have any temporary variables. However, it is questionable whether it is more efficient to take this into consideration or not.

A closure is used to represent an unexecuted literal block which was specified as an argument to a method. Within the called method, it may be required that the literal block argument (a closure) be evaluated, in which case the called method evaluates the closure object argument (via a message send to some object, usually the closure itself, that knows how to execute closure objects).

3.3.5. MSReturnObject

An MSReturnObject is a special wrapper that contains an MSObject. Consider a method block that contains a literal block. If an explicit return is executed in the literal block, then control must be returned to the caller of the method block. Note that the literal block may have been passed as an argument to an arbitrary number of methods before it was actually executed. The literal block returns an MSReturnObject to its caller. The MSReturnObject contains the actual return object as well as the context of the method in which the literal block lexically appears. When the caller receives an MSReturnObject, it checks to see if the current context (the context on the top of the dynamic context chain) is equal to the context stored within the MSReturnObject. If it is equal, the caller strips the MSReturnObject wrapper from the actual MSObject and returns the MSObject. If it isn't equal, it simply returns the MSReturnObject to its caller. The dynamic context stack will have changed, and the process is repeated until a match is found and the real MSObject is returned.

As a contrived example (Figure 2), suppose the method *item:block:* is defined for a particular class Example. Senders of this message specify a literal block as the second argument. Suppose in addition that this method sends that literal block argument to another method called *properBlockEval:*, and only within this message is the block actually evaluated (via the *value* method). If the literal block contains an explicit return, that method immediately returns the explicit value. Furthermore, *item:block:* immediately returns to its sender with that return value. Finally, the sending block returns with the specified value.

The result of performing *Example new example* is the integer '1'. Note that the explicit return of the integer 2 within the method *example* does not get executed. Furthermore, the method *incrBlockEval* within *properBlockEval:* is also not executed.

```
example ->
[
  self item: #primary block: [^1 ].
  ^2
]

item:block ->
[ :item :block |
   | tmp |
   item isPrimary
      ifTrue: [ self properBlockEval: block. ]
      ifFalse: [ ^nil ]
]

properBlockEval: ->
[ :block |
   self incrBlockCall.
   block value.
   self incrBlockEval.
]
```

Figure 2 : Explicit Return Behavior

3.4. Program Classes

Some utility classes are used for parsing. These include a scanner (MSScanner), a programming environment (MSPEnv), and an inheritance conflict resolution class (MSConflictSet).

3.4.1. Programming Environment — MSPEnv

A single instance of the ST80 class MSPEnv represents an entire MS programming environment, performing similar duties to the System Browser in ST80, plus much more. It provides facilities for parsing new MS programs into internal representations, as well as browsing and manipulating the internal representation of MS code existing in the program. It also maintains the method cache tables, assigns unique identifying indices to classes, maintains class index-to-class maps, maintains the uniqueness of syntactically specified literals used in the program, maintains code for primitives as well as facilities for defining them and mapping them to appropriate MS classes, monitors modifications to hard coded modules, classes and methods, and controls the incremental coloring and re-partitioning needed to provide efficient method dispatch[†].

[†] Because the topic is so important to this implementation of MS, a discussion of

The state of an MSPEnv includes : *primitives*, *library*, *scanner*, *display*, *literals*, *numLiterals*, *instTable*, *classTable*, *classMap*, *classInfo* and *numClasses*.

3.4.1.1. Browsing and Defining Modules

An instance of MSPEnv maintains a dictionary (*library*) associating strings representing MS module names with the internal representations of the modules, for all modules parsed within the environment. It also provides facilities for displaying modules, all classes within a module, all methods within a class, and the method code itself. This is similar to the System Browser in ST80. The current implementation is simplistic, and a substantial amount of work can be done to improve it.

Text files representing MS modules can be brought into the programming environment by sending a MSPEnv instance the message *parse:* with a string argument representing a UNIX file. The string represented by the file is converted into an internal representation (parse-tree) and stored within the MSPEnv. A discussion of parsing and the internal representation is presented later in the chapter.

3.4.1.2. Literals

Within Modular Smalltalk, literals are required to be immutable. This implies that a literal string assigned to a variable cannot have any of its character changed. One technique of guaranteeing immutability is to make literals unique. This is advantageous not only from a space saving perspective, but also with respect to the object identity method (==). If literals were not unique, message sends such as *1==1* or *"hello"=="hello"* would evaluate to false under the normal implementation which comparies two pointers for equality. However, there is some runtime overhead involved in enforcing unique literals.

The runtime efficiency problem occurs because uniqueness of literals suggests that all integer, float and character objects should be unique. This in turn implies that the collection of literals specified syntactically within a program is not necessarily the entirety of literals

---

the incremental coloring algorithm and method dispatch is reserved until Ch. 4.

objects used at runtime, because every instance of *Integer*, *Float* and *Character* should be a literal, and it is possible for an instance of one of these classes to be created dynamically at run-time without being statically specified as a literal. However, this results in a potentially infinite number of literals, and thus they cannot all be generated before execution of the program. This requires that each time a new integer/float/character instance is created at run-time, a check be made to see if that literal instance already exists (in a global literal hash table). If it does, the existing object is returned; otherwise a new instance is created, stored in the literal hash table, and returned. Even though hash tables could make the searching fairly efficient, this does put a severe limit on the efficiency of both integer and floating point operations within the language. Because of this, the uniqueness of literals may not be strictly enforced. However, this problem does not preclude the attempt to keep the number of objects representing the same literal to an efficient minimum. Literals specified syntactically in the program should be represented only once. Literals created during runtime may or may not be unique, which implies that the == method should be written to handle this possiblity. Note that the unique literal approach is not necessarily impossible. Most intensive number usage involves the use of looping constructs, such as the *to:do:* method. It is possible to avoid generating a new MS literal object to represent every number in the specified range by writing the *to:do:* method as a primitive. Within the primitive, one instance of the appropriate class (Integer,Float, etc.) is created. This object stores within its state the representation of the number within the base language. Thus, this base number can be changed on successive iterations of the loop without having to create an entire new object each time. Thus, the only mathematical operations which will necessarily introduce inefficiencies are those operations which are not primitives.

3.4.1.3. Primitives

In ST80, primitives are specified by unique integers. However, in MS primitives need not have an associated integer, since the defining class and method selector uniquely identify a primitive. The MSPEnv is responsible for maintaining a mapping from class-selector pairs to appropriate primitive implementations within the base language. Obviously, it is the responsibility of the programmer defining a primitive to implement the primitive in the base language and to notify the PE of the method to use. Since Smalltalk is used as a base language for the PE and C is used for stand-alone execution, the primitive behavior must be implemented in both languages for each MS primitive declared.

3.4.1.4. Kernel module and Kernel group

There exists a special module called the *Kernel* module, in which required classes and methods are specified. Because the required classes can potentially have superclasses (even though such super classes are not required), a concept referred to as the *Kernel Group* is introduced. The Kernel Group consists of the Kernel module and all modules recursively imported by it. One of the design goals was to keep the amount of special treatment of the Kernel module (and Kernel group) to a minimum. One subgoal that was deemed crucial was that the parsing of the Kernel Group be done by the same parsing methods as for any other MS Module. Hand compilation of the kernel group would have been far too restrictive.

Some special case code is required before parsing occurs because certain classes and objects are needed during parsing. These classes and objects include all of the literal (required) classes (Integer, Float, Character, String, MethodSelector, Array) since literals imply the creation of an object of a specific class at the time of parsing. Furthermore, the pseudo-variables *nil*, *true*, and *false* imply the need for unique instances of classes UndefinedObject, True and False respectively. One problem with the language specification is that the classes True and False are not required, nor is there any official mention for the requirement of unique instances of them (true and false). Because a language is useless with selection, these classes and pseudo-variables should be part of the language specification.

Since only the objects themselves are required (the state and behavior of these objects is not needed during parsing) a simple solution to the problem exists. Each time a new MSPEnv instance is made, MSObjects representing the required classes are made. However, no state or behavior is given to these created objects. The MSPEnv keeps a record of which instance and class objects have been created. Thus, during parsing, if a class object or instance object is being defined which has already been created, the specified information is associated with the created object instead of creating a new object. This allows the objects to exist before they are parsed, but leaves the specification of what those objects contain in the realm of MS programs.

3.4.1.5. Class Information

In the compiler, when C code is to be generated, it is much more efficient to identify classes by a unique index than by their associated name. Thus, the environment is responsible for assigning indices to classes and maintaining a classIndex-to-class map. Each time a MSCClass parse node (see below for details) is encountered during parsing, one of its responsibilities is to register the class with the environment.

3.4.1.6. The Cache Tables

The environment maintains two cache tables, one for instance behavior (*instTable*), and one for class behavior (*classTable*). A full discussion of the state and behavior of cache tables is discussed later. Each time a new class or selector is parsed, the node class doing the parsing must notify the appropriate table within the environment. Thus, if instance behavior is being parsed, and a new class definition is encountered, the MSCClass parse node must send the *addSuperClasses:toClass:usingConflict:* method to the instTable of the environment. This message is sent immediately upon establishing the superclasses of the class being parsed. The argument to *usingConflict:* is expected to be an instance of MSConflictSet, which will be described below. When a new selector is being parsed, the message *forceSelector:forClass:method:behavior:* is sent.

3.4.1.7. Miscellaneous Environment State

A variety of miscellaneous instances variables are used within the environment to maintain the current class being parsed (*currentParsingClass*), the current block being parsed (*currentParsingBlock*), the level of the module relevant to the current level (*moduleLevel*) and the number of literal blocks encountered at the current level (*blockCount*).

3.4.2. Scanner

An instance of MSScanner is responsible for converting an MS source string into tokens. Tokens are represented by special classes, called *token classes*, which will be described in section 3.5. A scanner knows the string to be parsed and the current parsing position within the string. The primary behavior of the scanner involves scanning the input string for the next token and advancing the string position past this token. In addition, the scanner has the capability to peek at subsequent tokens (obtaining the token without advancing the string position). For MS, a look-ahead of more than two tokens is never required.

Since tokens are reprented by instances of token classes, it is the scanner's responsibility to create such instances and return them. Briefly, the scanner knows the sequence of characters that represent any given token. The character at the current string position (and possible subsequent characters) determines which token is being represented. The scanner creates an instance of the appropriate token class, initializes the token's state as needed, and returns it to the caller. What is assigned for token state depends on the kind of token. There are two different kinds : *variable* and *constant*. Keywords and static syntactic constructs within the language are *constant tokens*, in that their state does not change. For example, there is a token (MSDeclarer) to represent '->' within MS. The value of this token is always the string '->'. On the other hand, *variable tokens* have a well-defined structure, but there may be an arbitrary number of actual tokens of that type. For example, there exists one token class to represent any integer, but the state of a given instance depends on the exact integer found on the input string.

The scanner is also responsible for error handling. If a syntax error occurs either during the tokenization process, or within a node parse, a message is sent to the scanner, who reports the error to the user and provides facilities for correcting the error so that the program segment can be reparsed.

3.4.3. MSCacheTable

Each MS environment contains two instances of MSCacheTable, one for instance behavior, and the other for class behavior. Each MSCacheTable has the following state. The *cache* state is an instance of an ST-80 Matrix class with behavior to easily add and modify existing entries and to expand the size of the matrix. Two dimensions are needed to store the relevant information, one for classes, the other for selectors (or something equivalent to selectors). In addition to the cache state, each MSCacheTable maintains information on all selectors encountered in the program within its realm, namely a mapping from MS message selector to corresponding MSSelector instance and a mapping from selector index to associated MSSelector instance. There is a unique instance of MSSelector for every syntactically unique MS message selector. The state and behavior of MSSelectors is described in Chapter 5.

3.4.4. MSConflictSet

An instance of MSConfictSet is created each time an instance or class behavior within a class declaration is encountered during parsing. It is responsible for maintaining a conflict set of all differing methods defined for a single selector.

3.5. Internal Representation

Figure 3 and Figure 4 represent two different approaches to compiling a program and generating base language code. The convential approach, shown in Figure 3, has a scanner which tokenizes an input stream, a parser which uses these tokens to create parse trees, and a code generator which uses the parse trees to produce base language code.

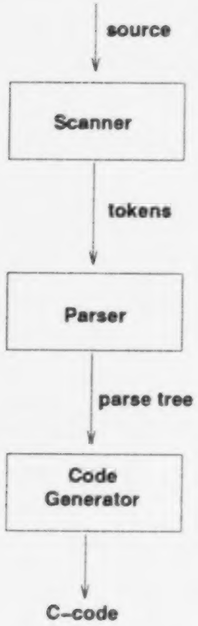

Figure 3 : Convential parsing/code generation

Figure 4, on the other hand, represents a profoundly different approach using the object-oriented paradigm. In this approach, tokens are still generated by a scanner, but this is the only similarity in design. Here, there is no single block of code to perform parsing or base language code generation. Instead, the BNF grammar describing a language is used to create a collection of *token classes* and a hierarchial collection of special *node classes* which form a parse tree. Each BNF grammar rule uniquely specifies a node class and the state of that node class. Since a BNF grammar rule consists of tokens and other BNF grammar rules, the state of a node class consists of tokens and other node classes. The fundamental difference of this approach is that each node class provides individualized behavior for a variety of activities, including parsing, execution, and decompilation.

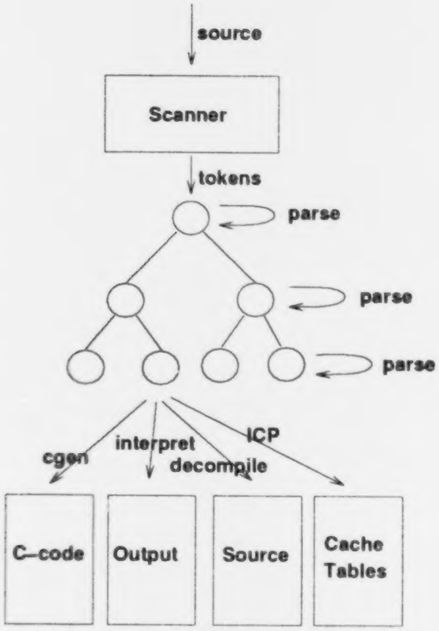

Figure 4 : Object oriented parsing/code generation

The sections below describe the ST-80 node classes and token classes that represent program fragments. The string representing such a program fragment is not treated as a sequence of characters, but rather as a sequence of tokens. These tokens are atomic constructs representing the smallest units of syntactically correct code. The rules in a BNF specification do not distinguish between tokens and nodes, and the decision as to where to stop tokenization and begin node parsing is sometimes arbitrary. Within MS, all BNF rules representing literals, syntactically constant language keywords and identifiers are treated as tokens, and all other rules are nodes classes.

In general, the exact number of rules in a BNF grammar specification is arbitrary, depending on the level of abstraction desired. Since BNF grammar rules are analgous to node classes, the BNF grammar specification should be rewritten so all subrules within a grammar rule are also node classes or token classes. Any subrules which are not to be represented as such should be placed directly into the specification for the rule being defined, instead of referred as a subrule. After rewriting, each grammar rule is to represent a node class. Thus, each grammar rule should be a syntactic construct within the language, and should also be a construct with some well defined execution (run-time) semantics.

The reason for requiring node classes to have execution semantics demonstrates the true power of the object-oriented approach to parsing. If each node class has both a syntactic (grammar) structure and an execution semantics, each node class can define behavior to parse itself *and* execute itself in an interpreted environment (the PE). In this way, one data structure (the node class) has multiple capabilities. As will be seen later, each node class has additional behavior beyond parsing and execution.

To see what type of rewriting of BNF rules is implied, compare Figures 5 and 6 with the grammar specifications in the second part of Appendix A. The first part of Appendix A provides an augmented BNF grammar similiar to Figures 5 and 6 that lists all information necessary to automate the node class state and parsing behavior process [SH93]. The second part of the appendix gives the BNF grammar rules as described in [Tex89]. Figures 5 and 6 show the rewritten BNF grammar structure so that node classes have well-defined execution semantics.

Within the BNF grammars below, expressions enclosed in square brackets ([...]) may occur zero or one times. Expressions enclosed in curly brackets ({...}) may occur zero or more times. All characters or strings (i.e. constant tokens) of characters appearing in the language are enclosed in single quotes ('...'). Nonprinting ASCII characters are given by using the associated C language backslash convention (i.e. for tab).

Tokens are denoted by >TokenName< and node classes are denoted by <NodeName>.

3.5.1. Token Classes

Every token class has a single named instance variable *value* which stores the value represented by the token. There are two categories of tokens, constant and variable. The value associated with constant token classes is always the same, whereas the value associated with variable token classes may be different between different instances of the token.

```
>Integer<             ::= ['-'] <digits> | <radix specifications> ['-'] <based digit> { <based digit>}
>LiteralSelector<     ::=  '#' <method selector>
>Float<               ::= ['-'] <digits> <sub floating point>
>Character<           ::=  '$' <printing character>
>Array<               ::=  '#(' {<literal>} ')'
>String<              ::=  ''' { <literal character> | <separator character> |  '"quote""quote"' | '""'  } '''
>KeywordSelector<     ::= <identifier> ':' { <identifier> ':' }
>BinarySelector<      ::= <operator character> {<operator character>}
>Declarer<            ::=  '->'
>VisPublic<           ::=  '(public)'
>VisPrivate<          ::=  '(private)'
>TypeVariable<        ::=  'variable'
>TypeBinary<          ::=  'binary'
>Word<                ::= <letter> { <letter> | <digit> }
```

Figure 5 : BNF Grammar Rules for Parse Tree Token Classes

3.5.1.1. Constant Token Classes

The constant token classes are : Declarer, VisPublic, VISPrivate, TypeVariable, TypeBinary. They represent the strings '->', '(public)', '(private)', 'variable', and 'binary' respectively.

3.5.1.2. Variable Token Classes

The variable token classes are : Integer, LiteralSelector, Float, Character, String, Array, KeywordSelector, BinarySelector, Word. The value state of instances of such token classes is always an ST-80 String, with the exception of Array, whose state is an Array of Strings. Conversion to appropriate internal representations is not the responsibility of the token class. Instead, there exists a node class Literal which defines behavior to provide such conversions, as well as enforcing uniqueness of literals within the PE.

3.5.2. Node Classes

Since node classes correspond to BNF grammar rules, each node class should know what tokens and grammar rules make up instances of themselves. This information allows behavior to be associated with each node class to parse itself. This is the fundamental difference between the conventional and object-oriented approach to parsing. In the former, one dedicated program is responsible for creating a parse-tree, whereas in the object-oriented paradigm, individual nodes within a template parse-tree (the node-classes) know how to initialize instances of themselves.

```
<Module>            ::= '{' 'module' [>String<] {<Binding>} '}'
<Binding>           ::= >Word< [>VisAttr<] '->' (<Import> | <ClassDefn> | <ModExpr>)
<Import>            ::= '{' [ 'import' >Word< ] 'from' >String< '}'
<ModExpr>           ::= '{' 'expression' <MessageSend> '}'
<ClassDefn>         ::= '{' 'class' [ '{' 'refines' { >Word< } '}' ] [ 'instance' <Behavior> ] [ 'class' <Behavior> ] '}'
<Behavior>          ::= '{' 'behavior' { <StateMethod> | <Method> } '}'
<StateMethod>       ::= '{' ( ( >UnarySelector< [>VisAttr<] >SingleKeywordSelector< [>VisAttr<] ) |
                        >SingleKeywordSelector< [>VisAttr<] >UnarySelector< [>VisAttr<] ) '}' '->' 'variable' ) | (
                        ( >UnarySelector< [>VisAttr<] >SingleKeywordSelector< [>VisAttr<] |
                        >SingleKeywordSelector< [>VisAttr<] >UnarySelector< [>VisAttr<] ) '|' (
                        >SingleKeywordSelector< [>VisAttr<] >DoubleKeywordSelector< [>VisAttr<] | >Doub-
                        leKeywordSelector< [>VisAttr<] >SingleKeywordSelector< [>VisAttr<] ) '}' '->' ( 'variable'
                        | 'binary' ) ) )
<Method>            ::= >Selector< [ >VisAttr< ] '->' ( [ 'method' ] ( <Block> | <PrimitiveMethod> | <UndefMethod> |
                        <AbstractMethod> | <AliasMethod> )
<AliasMethod>       ::= alias' >Word< >Selector<
<PrimitiveMethod>   ::= 'primitive'
<AbstractMethod>    ::= 'abstract'
<UndefMethod>       ::= 'undefined'
<Block>             ::= '[' [ { ':' >Word< } '|' ] [ '|' { >Word< } '|' ] { ( <Assignment> | <MessageSend> ) '.' } [ ( (
                        <Assignment> | <MessageSend> ) | '^' ( <Assignment> | <MessageSend> ) ) [ '.' ] ]
<Assignment>        ::= [ >Word(addVariable)< ':=' ] <MessageSend>
<MessageSend>       ::= ( >Word< | <Literal> | <Block> | <Expression> ) [ <Message> { ';' <Message> } ]
<Message>           ::= very complex
<Expression>        ::= '(' ( <Assignment> | <MessageSend> ) ')'
<Literal>           ::= >Literal<
```

Figure 6 : BNF Grammar Rules for Parse Tree Node Classes

Each node class has a *parseWith:* message selector defined. The behavior associated with this message depends entirely on the BNF grammar associated with the node class. The argument to the message is an instance of a scanner, which is used to obtain tokens from the input stream. The method knows what tokens are expected from the input, and must simply read tokens from input and compare them against what is expected. During this process, the method will eventually encounter a token that is the start of a grammar subrule. Since this grammar subrule is also represented by a node class, the method creates an instance of the appropriate node class and has this new instance parse itself. After the newly created node class instance has finished parsing it returns itself and the method can add this node class instance to its own state.

Figure 7 and Figure 8 show the has-a hierarchy of the node classes. Ovals refer to nodes and squares refer to tokens which must be stored as part of a node class' state. Constant tokens which are expected to exist on the input stream but which need not be stored are not shown in the figures. Single solid lines indicate that the node class on the top contains the node class/token on the bottom. Single dotted lines indicate that the has-a relationship is optional. Double solid lines indicate that one or more instances of the node class/token can be contained within the parent node class, and double dotted lines indicate zero or more instances.

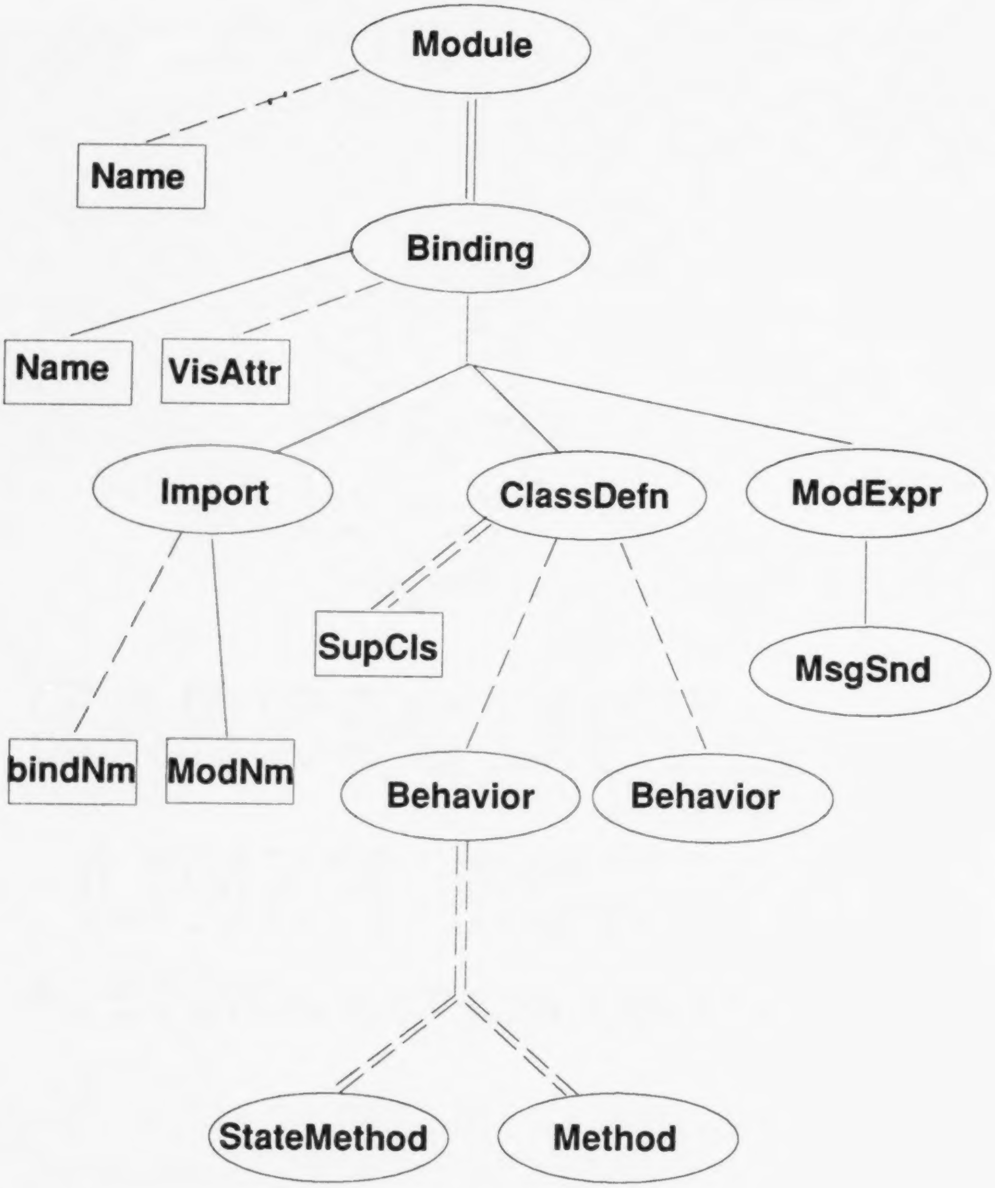
Module
Name
Binding
Name
VisAttr
Import
ClassDefn
ModExpr
SupCls
MsgSnd
bindNm
ModNm
Behavior
Behavior
StateMethod
Method

Figure 7 : High-Level Node Class Relationships

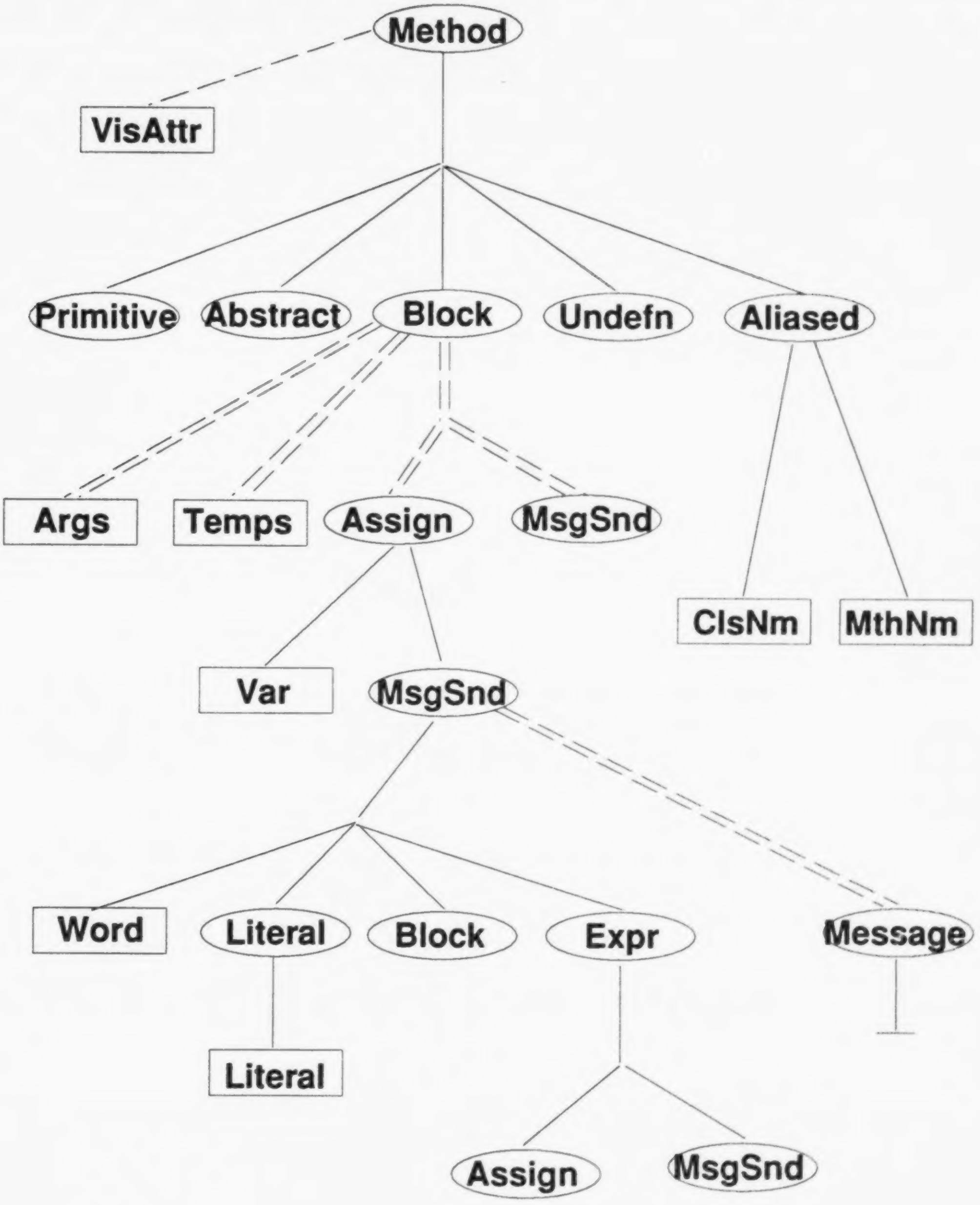
Method
VisAttr
Primitive
Abstract
Block
Undefn
Aliased
Args
Temps
Assign
MsgSnd
ClsNm
MthNm
Var
MsgSnd
Word
Literal
Block
Expr
Message
Literal
Assign
MsgSnd

Figure 8 : Low-Level Node Class Relationships

One of the goals during the PE parser/interpreter implementation was to keep the run-time and compile-time responsibilities completely separate, thus providing for flexible implementations. To this end, there are actually two node classes for every BNF rule, a run-time node class and a compile-time node class. During parsing, the state of the compile-time node class must maintain all information needed to recreate the source program (decompile). However, only part of the information stored at compile-time is needed during execution. Thus, the compile-time node class is made a sub-class of the run-time class. The run-time class defines only that state needed during execution, and the compile-time class inherits this state, and adds any additional state needed. The run-time class is responsible for implementing behavior to perform execution of the parse tree, and the compile-time class must implement additional behavior to parse, de-compile, and perform C-code generation.

During the description of individual node classes, references to an *expression node class* refer to any one of : MSName, MSLiteral, MSMessageSend, MSCascadedMessageSend, MSAssignment or MSReturn.

The MS code in Figure 9 expands on the module defined in Chapter 2. Figures 5 through 9 should be kept in mind while reading the following sections.

```
{
    module 'Numbers'                                        "Module Start"

    Object -> { from 'Kernel' }                             "NameImport Declaration"

    { use 'MathConstants' }                                 "ModuleImport Declaration"

    Complex (public) ->                                     "Class Declaration Start"
    [
        class { refines Object }                            "Superclass Declaration"

        instance                                            "Instance Behavior Start"
        { behavior
            { real real: } -> variable                      "State Method Declaration"
            { imag imag: } -> variable

            * -> primitive                                  "Primitive Method Declaration"
            < -> undefined                                  "Undefined Method Declaration"
            dummy -> abstract                               "Abstract Method Declaration"
            isNew -> alias Object isNew                     "Alias Method Declaration"

            + -> method                                     "Binary Block Method Start"
            [ :aComplex |                                   "Formal Paramaters"
                | result |                                  "Temporary Variable"

                result := Complex                           "Assignment"
                        real: (self real + aComplex real)   "Unary & Binary Message Send"
                        imag: (self imag + aComplex imag).  "Keyword Message Send"
                ^result                                     "Explicit Return"
            ]                                               "Binary Block Method End"

            asReal ->                                       "Method with Literal Blocks"
            [
                self imag == 0
                    ifTrue: [ ^self real ]                  "Literal Block #1"
                    ifFalse: [ ^nil ].                      "Literal Block #2"
            ]
        }                                                   "Instance Behavior End"

        class                                               "Class Behavior Start"
        { behavior
            real:imag: ->                                   "Keyword Block Method Start"
            [ :real :imag |                                 "Method Arguments"
                self new                                    "Unary Message Send"
                    real: real; imag: imag.                 "Cascaded Message Send"
            ]                                               "Keyword Block Method End"
        }                                                   "Class Behavior End"
    ]                                                       "Class Declaration End"

    i -> { expression Complex real: 0 imag: 1 }             "Module Expression"
}                                                           "Module End"
```

Figure 9 : MS Module Code — Extended Numbers Module

3.5.2.1. MSName

An MS variable is represented by an instance of class MSName. Variables are mapped to level-offset pairs specifying a unique location within a context chain. The value of a variable is stored in a location uniquely identifed by the level and offset associated with it.

The MSName class has state for a level and offset. The associated compiler class has additional state for the identifier represented by this level-offset pair. It is used during decompilation to print out the variable being represented.

As an example, Figure 9 defines many variables in the module. Remember that the level of a variable is relative to the current level. Thus, within a literal block within a method (Figure 9: *Literal Block #1 or Literal Block #2*) the module context has level 2. The first variable within the module's scope is 'Object', with offset 0. Thus, a level-offset pair of 2-0 uniquely identifies the position within the static context chain of the object represented by the class object 'Complex' (when within one of the aforementioned literal blocks). Because every context representing a block method or literal block stores the receiver of the method in the first context location, references to the pseudovariable *self* can always be obtained by the level-offset pair 0-0. If the current context is a literal block, *self* could also be accessed by 1-0, etc. Although there is a degree of space inefficiency in this implementation, it does give optimal execution performance. If *self* was stored only in the context of block methods (i.e. not in literal block contexts), then the time needed to access the 'self' variable would be dependent on how deeply imbedded within a block one was.

3.5.2.2. MSLiteral

Although there is a token class that represents literals, the current implementation also has a node class for them. Furthermore, in the current implementation MSLiteral is a subclass of MSObject. Subsequent versions will not have a node class to represent literals, but some language class which provides the current behaviors of MSLiteral will be implemented. The behaviors provided include conversion of a token value string to an appropriate internal form (i.e. the string representing a floating number is converted to a Float) and a clean method of registering literals with the environment to insure their uniqueness.

As an example of the current use of MSLiteral node clases, the literal tokens 0 and 1

(Figure 9: *Module Expression*), upon being encountered during the parsing of a message send, are used to create instances of MSLiteral whose state consists of a unique representation of the literal in question. The creation message asks the environment (MSPEnv) if it has a representation for the given token already. If it does, that object is used, otherwise an appropriate MSObject is created and registered with the environment, and this new object is used.

3.5.2.3. MSMessageSend

Syntactically, a message send contains a receiver, a selector, and zero or more arguments. However, the state for the node class MSMessageSend stores only the receiver and an instance of the node class MSMessage. The receiver is an instance of any MS expression node class. An MSMessageSend also maintains the defining block node class instance (i.e. the instance of MSBlock representing the entire method where the message send syntactically occurs). If the message send occurs in a module expression, the defining block is the appropriate MSModule node class.

Instances of MSMessage store an array of the arguments to the message and the selector name. The receiver-message pair representation of MSMessageSend is used instead of placing the receiver, selector and arguments all within a single node class because it simplifies support for cascaded messages sends, which have a single receiver and multiple selector-arg pairs. An MSCascadedMessageSend is similar to an MSMessageSend except that it contains an array of MSMessages instead of just one. The compiler node classes do not add any extra state.

Figure 9 gives many examples of message sends. *asReal* (Figure 9: *Method with Literal Blocks*) contains a keyword message (ifTrue:ifFalse:) whose receiver is the result of a binary message send (==) whose receiver is the result of a unary message send (imag) whose receiver is self. For each of these nested message sends, an instance of MSMessageSend would hold the receiver and an MSMessage, which would hold the appropriate selector and any arguments. *real:imag:* (Figure 9: *Keyword Block Method Start*) contains an example of a cascaded message send, where the receiver is the result of *self new* and the two message sends cascaded are *real:* and *imag:*.

3.5.2.4. MSReturn

Syntactically, a return statement consists of the literal token '^' followed by any MS expression node class. MSReturn state consists solely of an expression node class. No compiler state is needed.

The runtime execution of an explicit return results in the evaluated form of the associated expression being returned as the result of the block method in which the return was found. This occurs even if the return was nested within many literal blocks. To provide this outcome, the execution behavior of instances of the MSReturn node class consists of encapsulating itself within an instance of a MSReturnObject wrapper. See the discussion below under MSBlock for more details.

Figure 9 has explicit returns within a block method (Figure 9: *Explicit Return*) and within a literal block (Figure 9: *Literal Block #1, Literal Block #2*).

3.5.2.5. MSAssignment

An assignment consists of an MSName to represent the assignment variable and an MS expression node class describing what is to be assigned to the variable. During evaluation of an assignment, the evaluated form of the expression is to be stored in the appropriate context location identified by the assignment variable. The object resulting from an assignment expression is the object assigned to the variable. This is important because the language requires that nested assignment statements be legal.

As an example (Figure 9: *Assignment*), the temporary variable *result* is assigned the result of a keyword message send.

3.5.2.6. MSBlock

Syntactically, a block contains lists for formal arguments, temporary variables and message expressions (statements). The only information needed at run-time is the number of arguments and temporaries, and the list of statements.

The compile-time node maintains dictionaries mapping variable names to their corresponding MSName instances. The variable names are needed for decompilation, and MSName instances are needed during compilation to enforce the language specification constraint that only temporary variables be assigned values. Thus, there are two separate dictionaries, one each for arguments and temporaries. These dictionaries contain not only the variables defined within the current scoping level, but also those defined in parent scoping levels. The block also maintains the MSClass node class instance in which the block syntactically exists, a flag denoting whether the block is literal or not, the name of the method, the name of the C function to be generated for compilation, and a list of all immediate subblocks of the current block. The subblocks are stored as they are encountered so as to make C generation easier. Since these literal blocks are usually arguments to arbitrarily deeply nested message sends, and since C code for these blocks must be generated separately from the C code for the message sends themselves, this list of subblocks must be obtained either during parsing or during C code generation.

In Figure 9 there are three names in the module context, *Object*, *Complex* and *i*. When parsing the block method for + (Figure 9: *Binary Block Method Start — Binary Block Method End*), upon creation of a MSBlock node class instance (discussed below), its argument dictionary (initially empty) is augmented by adding copies of all MSName instances in the module's context (namely *Object*, *Complex* and *i*). Within the module context, the levels of the MSName instances is 0, but within the block method context, they are level 1. Thus, the copied MSName instances in the argDict have their level incremented by 1 (but their offset stays the same). The argument variables for the block (Figure 9: *Formal Parameters*) are then parsed and added to the argDict with level 0 and offsets starting at 0. Thus *aComplex* has level 0 and offset 0. Next, the temporary variables (Figure 9: *Temporary Variable*) are added with level 0 and offsets starting after the last offset from the argument list. Thus *result* is added to the *tempDict* state with level 0 and offset 1.

Continuing the parsing of the MSBlock instance, an MSAssignment node class instance is formed and added to the (currently empty) *statement* OrderedCollection. Then, an MSReturn instance is added after the MSAssignment instance (the order of statements is obviously important).

3.5.2.7. MSMethod

As discussed in Chapter 2, MS has six different types of method specification, MSStateMethod, MSPrimitiveMethod, MSAbstractMethod, MSUndefinedMethod, MSAliasedMethod, and MSBlock. The abstract superclass of all of these execept MSBlock is MSMethod, which has state for recording whether or not the given method's visibility is private. It also defines behavior common to all method node class instances, such as initialization, default C code generation, testing methods and access methods for its state. MSBlock defines the state and behavior found in MSMethod so that all six node classes are compatible.

The compiler node classes for each of the message node classes also maintains the defining class of the method, the MSClass node class in which the definition for the method syntactically resides. It is used during execution of private methods to insure that the class of the receiver is the same as the defining class of the method. It is also used during inheritance conflict resolution to determine the 'sameness' of methods. However, because the PE implementation is in ST-80, which has single inheritance, and because compiler classes are already subclasses of their associated runtime node class, there is no way of abstracting this state (and associated behavior). Thus, in addition to the state and behavior described below, each compiler method node class must explicitly define this state.

3.5.2.7.1. MSStateMethod

In the current implementation, the MSStateMethod class is an abstract superclass maintaining a state index. There are two or four state methods associated with a given state method specification. The state index represents which specification an individual state method belongs to. State indices start at 1 with the first syntactically encountered state method declaration. MSStateMethod has 4 subclasses, MSSizeAccess, MSSizeChange, MSAccess and MSChange, which in turn are abstract superclasses for 2 (MSByteIdxSizeAccess, MSIdxSizeAccess), 2 (MSByteIdxSizeChange, MSIdxSizeChange), 3 (MSByteIdxAccess, MSIdxAccess, MSNamedAccess) and 3 (MSByteIdxChange, MSIdxChange, MSNamedChange) classes respectively. Each message selector in a state method specification is represented by a node class instance.

In Figure 9, the Complex class declares state methods for accessing and changing the real and imaginary components of the number (Figure 9: *State Method Declarations*). The access methods are real and *real:*, and the changing methods are *real:* and *imag:*. An example of indexed and byte indexed state method specification can be found in Appendix D in the definitions of *Array* and *String* in the Kernel module.

3.5.2.7.2. MSPrimitive, MSAbstract, MSUndefined

Instances of MSPrimitive record the selector name of the associated ST-80 primitive, and a string denoting the name of the primitive within the compiled environment.

Within MS, the syntax of declaring a method primitive is trivial (Figure 9: *Primitive Method Declaration*). However it is the responsibility of the programmer to insure that the MSPEnv in which a module was being parsed has an appropriate ST80 selector and C function name associated with the class-selector pair identifying a primitive. The programmer is also reponsible for writing the ST80 code (within the MSPEnv class) and the C code (in the back-end C library) using the names specified.

MSAbstractMethod does not have any additional state. Its runtime execution results in an MS error message similar (currently identical, in the PE) to the ST-80 message *subclassResponsibility*. Attempting to execute an abstract method (Figure 9: *Abstract Method Declaration*) will generate the aforementioned MS error.

MSCUndefinedMethod also doesn't have any special state (Figure 9: *Undefined Method*). Its runtime execution results in an MS error message similar (currently identical, in the PE) to the ST-80 message *shouldNotImplement*.

3.5.2.7.3. MSAliasedMethod

An MSAliasedMethods stores the MSClass node class from which the alias is to take place, and the MSSelector which is to be aliased (Figure 9: *Alias Method Declaration*). The aliased class must be an immediate superclass of the class currently being parsed, and this condition is checked for during the parsing of an aliased method declaration. If this condition fails, a compiler error is generated. Furthermore, the selector must be the name of a method within the aliased class, although this is not checked for until runtime.

The runtime execution of an instance of MSAliasedMethod is simply the execution of its associated alias method.

3.5.2.8. MSClass

Syntactically, a class (Figure 9: *Class Declaration Start — Class Declaration End*) consists of a list of superclasses (Figure 9: *Superclass Declaration*), an instance behavior (Figure 9: *Instance Behavior Start — Instance Behavior End*), and a class behavior (Figure 9: *Class Behavior Start — Class Behavior End*). Required run-time information consists of the number of named, indexed and byte indexed states for the instance behavior. In addition, if a naive recursive look-up method dispatch strategy is used, then a map of selectors to methods and visibility attributes is required for instances and for classes. The equivalent compile-time class stores a mapping from selectors to methods for both instance and class behaviors.

3.5.2.9. MSModule

At run-time, an MSModule maintains a context of name-value pairs. An MS module can bind a name to instances of one of five node classes, MSClass, MSNameImport, MSModuleImport, MSClassExtension, or MSModuleExpression. Thus, MSModule instances have state for an array of bindings and a parallel array for the context, whose indicies store the evaluated versions of the corresponding index in the bindings array.

An MSModuleExpression is a wrapper than contains an instance of one of MSMessageSend, MSCascadedMessageSend, MSName or MSLiteral. Instances of MSReturn, MSBlock or MSAssignment are not allowed, although literal blocks can exist as arguments. Thus, although the language specification allows explicit returns to exist within literal blocks within module expressions, the effect of this is not defined. To maintain consistency with the corresponding action within blocks, the entire module should stop immediately. The utility of such an action is doubtful, and thus explicit returns should not be allowed within module expressions and should result in a parsing error. However, the current implementation does not check for this. Module expressions are the entry point of meaningful execution within a program (Figure 9: *Module Expression*).

An MSNameImport represents the binding of a name within the current module to the object represented by a name imported from another module's context (Figure 9: *NameImport Declaration*). It thus has state for an MSName instance whose offset specifies the index within the source module's context, and whose level is unused (and always 0), as well as state for an MSModule instance from which to import. Since an exporting module is executed before a module that imports from it, the object will exist when it is referenced.

An MSModuleImport represents the importation of every binding from a specified module (Figure 9: *ModuleImport Declaration*). It is simply syntactic shorthand for many MSNameImport specifications. Its state consists solely of an MSModule instance from which to import. Such a node class does not currently exist. The current implementation actually generates MSNameImport node classes for every name in the source module. Although this works properly, decompilation does not give the expected results.

An MSClassExtension has state for an MSClass instance representing the class to be extended (or maybe only the name of the class, since it would otherwise be necessary for the base class to exists in order to parse a class extension). Also included is state for dictionary state analagous to that for MSClass. MSClassExtensions have not been implemented in the current version.

Chapter 4 : Method Dispatch

4.1. Method Dispatch Techniques

Method dispatch is needed because the object-oriented paradigm allows two distinct classes to define differing methods for the same selector. Thus a selector by itself does not necessarily uniquely identify a single executable method; usually the class of the receiver object is also needed. Since the receiver (and thus the class of the receiver) cannot be determined at compile-time, method dispatch is needed to resolve the issue at run-time. As will be seen in the discussion below, the selection of a method dispatch technique is usually a tradeoff between time and space [CU91].

4.1.1. Smalltalk's Lookup Dispatch

The basic ST-80 method dispatch algorithm involves a path traversal, starting in the method dictionary of the receiver of the message[GR89]. If no method for the message selector is found, the method dictionary of the superclass is checked. This process continues until a method is found or no further superclasses exist (at which time a *messageNotUnderstood* method is invoked to warn the user). This algorithm is slow for long inheritance chains and is even slower in MS where multiple inheritance is involved and multiple superclass chains need to be searched (resulting in a tree traversal).

Note that this lookup dispatch approach is as efficient as possible from a space perspective.

4.1.2. Cache-table Dispatch

The fastest possible implementation (assuming the needed for method dispatch) uses a two-dimensional cache-table which stores a method to use for each class-selector pair. Each selector in the environment would have a unique row in this table, and each class would have a unique column. At run-time, the receiver object class and selector are used to obtain an entry into the cachetable, which stores the method to execute. Although this approach is fast, it is not feasible since the space required is the product of the number of classes and selectors.

4.1.3. Colored Cache-table Dispatch

A similar cache-table algorithm [DMSV89] assigns colors to selectors, utilizing information about which classes understand specific selectors to reduce the amount of space needed. Each selector no longer has a unique row, but instead shares the row with other selectors that do not conflict with it. The index of the row to which a selector belongs is refered to as the selector's *color*. Two different selectors (*alpha* and *beta*) can be assigned the same color as long as the set of classes using *alpha* is disjoint from the set of classes using *beta*. For example, consider the case where there are 7 classes A-G. Assume that classes A, C and G recognize the message selector *alpha* but not *beta*. Classes B and E recognizes the message selector *beta*, but not *alpha*. Finally, classes D and F recognize neither. A single color can be used to represent the selectors *alpha* and *beta* (Table 3).

color	A	B	C	D	E	F	G
1	alpha	beta	alpha	empty	beta	empty	alpha

Table 3 : Colored CacheTable : Two selectors sharing same color

The run-time efficiency of this approach is only slightly less than in the non-colored cache-table approach described above. The inefficiency occurs because the selector and class are used to obtain a color-class entry into the cache-table. The cache-table must, in addition to storing the method, store the selector for which it is applicable. A run-time check for equality between the current selector and the store selector must be performed. If they are not equal, the entry should be treated as if it were empty (the generation of an appropriate message telling the user the specified method was not found).

This technique provides substantial space savings over the previous cache-table approach, at the expense of added compilation time. Each time a new class and/or selector is added to the environment, the current color of an arbitrary number of selectors can change. As discussed in [AR92], the technique proposed in [DMSV89] separates the processes of defining code and coloring the selectors. However, this means that the entire environment must be recolored each time compilation is desired. The resulting compilation time for large systems makes the technique useless.

4.1.4. Incremental Coloring Cache-table Dispatch

The paper [AR92] suggests a method of performing incremental coloring, in which the color of a selector is determined when it is defined. We will refer to this algorithm as the André-Royer algorithm. This algorithm is presented below and analyzed in detail.

4.1.5. Incremental Coloring and Partitioning Dispatch (ICP)

The method dispatch technique used for MS is an incremental coloring and partitioning (ICP) technique based on the André-Royer algorithm. ICP is more general than the André-Royer algorithm, which has been enhanced in a variety of ways. Both the André-Royer algorithm and the ICP algorithm are discussed in detail below.

The most important difference between the ICP dispatch algorithm used for MS and the André-Royer algorithm involves the maintenance of information which will allow for the efficient determination of whether method dispatch can be avoided entirely for certain message sends. Although the André-Royer algorithm does partition selectors into differing categories, these partition types are only used as abstractions while defining the algorithm. Furthermore, the André-Royer algorithm does not provide efficient means of obtaining these partition types, and thus the algorithm itself is inefficient. Within ICP, partition types are substantially more important and the algorithm maintains information which allows for immediate determination of a selector's partition type by incrementally adjusting partition information at the same time that color information is adjusted.

Partition types are important because selectors of certain partition types are uniquely specified by the selector itself, and thus no lookup is required so a method to be executed can be found at compile-time. However, the number of selectors which fall into one of these partition types is quite low without static typing. Thus, the full power of the ICP algorithm cannot be realized in MS without a language extension to provide for static typing.

4.2. André-Royer algorithm

Before presenting the ICP algorithm as implemented for MS, the André-Royer algorithm is given. Enhancements, corrections and generalizations are made to the algorithm. The final result of these modifications is the ICP algorithm.

Although the André-Royer algorithm does not explicitly state it, each entry in the cache-table consists of a method and a selector. Such an entry will be refered to as a *division* in subsequent discussions.

The definitions in the table below are needed in order to understand the algorithm. The letters C and C_i refer to classes, G to a group of classes, S to a selector, D and E to divisions, and L to a color. The symbol Ω is used to denote an empty division (i.e. empty cache-table entry). The notation $C_i < C$ means that class C_i is in the inheritance sub-graph with root class C.

Symbol	**Definition**
divisionAt[L, C]	the cache-table entry for color L and class C
divisionSelector(D)	the selector associated with division D
color(S)	color mapped to selector S
superclasses(C)	immediate superclasses of C
nativeBehavior(C)	set of all selectors explicitly defined in class C
superBehavior(C)	$= \cup_{C_i \in \text{superclasses}(C)} \text{completeBehavior}(C_i)$
exceptedBehavior(C)	$= \{ S \mid S \in \text{superBehavior}(C) \text{ and } S \notin \text{nativeBehavior}(C) \}$
completeBehavior(C)	$= (\text{nativeBehavior}(C) \cup \text{superBehavior}(C)) - \text{exceptedBehavior}(C)$
subBehavior(C)	$= \cup_{C_i \in \text{allSubclasses}(C)} \text{nativeBehavior}(C_i)$
classesDefiningSelector(S)	$= \{C \mid S \in \text{nativeBehavior}(C)\}$
allSubClasses(C)	$= \{C_i \mid C_i < C\}$
allSuperClasses(C)	$= \{C_i \mid C < C_i \}$
relatedClasses(C)	$= \text{allSubClasses}(C) \cup \text{allSuperClasses}(C) \cup \{C\}$
colorsUsedBy(C)	$= \{ L \mid \text{divisionAt}[L, C] \neq \Omega\}$
colorsFreeFor(G)	$= \cap_{C \in G} \{ L \mid \text{divisionAt}[L, C] \equiv \Omega\}$
classesUsingColor(L)	$= \{C \mid \text{divisionAt}[L, C] \neq \Omega\}$
dependentClasses(D)	$= \{C_i \mid C_i < C, C_j < C, S = \text{divisionSelector}(D), S \notin \text{nativeBehavior}(C_i)$ and if $S \in \text{nativeBehavior}(C_j)$ then $C_j < C_i\}$

Table 4 : Definitions for the André-Royer Algorithm

Partition	**Definition**
specific(S,C)	= [classesDefiningSelector(S) = {C}]
separate(S,C)	= [relatedClasses(C) $\cap$ classesDefiningSelector(S) = {C}]
redefined(S,C)	= [S $\in$ superBehavior(C)]
declared(S,C)	= [not specific(D) and not separate(D) and not redefined(D)]

Table 5 : André-Royer Partition Types for selectors

The colors assigned to selectors can change when a new class is added to the environment, or when a new method is associated with a selector in an existing class. Thus the original André-Royer algorithm is divided into two parts, one handling incremental coloring when a new class is added to the cache-table, and the other handling a new method. The André-Royer Class Algorithm has no flaws and is left unchanged within the ICP algorithm.

4.2.1. Actions During Class Definition

During compilation, upon encountering a declaration for a class, some steps must be taken before any methods within the class are parsed. Specifically, a new class column must be added to the cache-table

4.2.1.1. Inheritance Copying

To provide for the mechanism of inheritance, the class columns for all superclasses are collected. For each color, the corresponding entries in these columns are compared. Depending on the conflict resolution scheme specified for the language under development, actions will be taken to add a unique value to the corresponding color row in the new class column, report an error, or postpone copying of values until conflicts have been resolved. Method conflict resolution in Modular Smalltalk requires that if conflicting methods exist, one such method be explicitly selected by the user (via an explicit definition or an alias) Hence, if a conflict exists for a particular color index due to the same selector having different implementations in different superclasses, the Modular Smalltalk implementation does not copy any value to the new class index array until one unique method is associated with the selector. The MSConflictSet class handles this automatically upon receiving alias information which results in a unique method for a given selector.

4.2.2. Selector Color Conflicts

Although the inheritance method conflict resolution scheme will vary between languages, there are invariant concerns during class definition. Predominant among them is the possibility that two classes, say A and B, have different selectors stored at the same color (say alpha and beta with color 1). If class C were to inherit from both of them, alpha and beta would no longer be allowed to have the same color. If such a conflict is detected during inheritance copying, one of these selectors must be moved to a different color. In addition to changing the color of the associated selector, this involves moving every division defining the selector from the old color index to the new color index (maintaining the same class indices, of course). Figure 10 gives the algorithm to implement the discussion above.

```
ICPNewClass( in/out C : Class )
    add a new column to cachetable.
    for all colors L, divisionAt[L,C] = $\Omega$

    for each $C_s \in$ superClasses( C ) do
        for each $L \in$ colorsUsedBy( $C_s$ )
            Let D = divisionAt[L,C]
            if ( $D \neq \Omega$ ) and
             ( divisionSelector( D ) ==
             divisionSelector( divisionAt[L,$C_s$] ) )
                MoveSelectorToFreeColor( divisionSelector(D) )
            endif
            divisionAt[L,C] = divisionAt[L,$C_s$]
end ICPNewClass

Procedure MoveSelectorToFreeColor(S)
    G = classesDefiningSelector(S)
    select L from colorsFreeFor(G)
    set D = new MSEmptyDivision
    for $C_i$ in G
        set divisionAt[$C_i$, L] = divisionAt[$C_i$, color(S)]
        set divisionAt($C_i$, color(S)] = D
end MoveSelectorToFreeColor
```

Figure 10 : André-Royer Class Algorithm

4.2.3. Actions During Method Definition

The original André-Royer method algorithm is concerned with assigning a color to a selector S defined in a class C. The partition type of the selector determines what type of recoloring is necessary.

The algorithm is presented in Figure 11, after which is a collection of notes which explain the algorithm and point out areas where enhancements or corrections can be made. After these notes is the modified André-Royer, which is the starting point for the ICP algorithm.

```
Algorithm AR(in C : Class, in/out S : Selector, in/out T: CacheTable)

    let L = color(S)

    Switch partition of S
        case specific: find free color for allSubClasses(C)                                  (NOTE 1)
        case redefined: no change                                                            (NOTE 2)
        case separate:                                                                       (NOTE 3)
            If color(S) $\in$ colorsFreeFor({C}) then no change                              (NOTE 4)
            else color(S) = any color in colorsFreeFor(classesUsingColor(color(S)))          (NOTE 5)
        case declared:                                                                       (NOTE 6)
            If for X < C, color(S) $\in$ colorsFreeFor({X}) or S $\in$ nativeBehavior(X) then
                no change                                                                    (NOTE 7)
            else color(S) = any color in colorsFreeFor(classesUsingColor(color(S)))          (NOTE 8)

    If L $\neq$ color(S)                                                                     (NOTE 9)
        for $C_i$ $\in$ classesUsingColor(L)
            divisionAt[color(S), $C_i$] = divisionAt[L, $C_i$]
            divisionAt[L, $C_i$] = $\Omega$
```

Figure 11 : The André-Royer Algorithm

Notes

1. If only one class in the PE defines S, partition(S) is *specific*. The color associated with the selector can be any color that is not used by subclasses of C. However, the André-Royer Algorithm makes the assumption that there are no exceptions to inheritance, which is why the class C itself is not also checked. Supposing inheritance exceptions do not occur, it is actually sufficient to check only the leaf subclasses of C for free colors, since selector S will be in the leaf subclasses if they are

in any of the classes between a leaf and C. However, to handle inheritance exceptions, the algorithm must check class C and every subclass of C since a superclass may use selectors that a subclass does not.

2. According to André-Royer, if the selector S exists in the behavior inherited from superclasses of C, partition(S) is *redefined*. Furthermore, redefined selectors do not need their colors changed. Note that the definition of redefined partitions has been carefully worded. If S is defined in a super class but an inheritance exception occurs before class C so that S is not part of the behavior inherited from superclasses of C, then S will not be considered redefined.

3. If the only class within the forest containing C which defines S is C, then partition(S) = *separate* (the forest containing C is the set of all superclasses of C, all subclasses of C, and C itself).

4. If S is declared, André-Royer states that one need only check to insure that the current color of S is free for C. However, this is not true. It must also be free in certain subclasses of C, namely all subclasses of C which will share the same method for S as C. The collection of such classes has been termed the *dependendClasses(C)*. Since S is separate, by definition the dependentClasses(C) = allSubClasses(C). As an example illustrating why André-Royer is not sufficient, suppose class B inherits from class A, and a selector beta is defined in class B, with color of 1. If class A later defines alpha, it is not sufficient to ensure that color 1 is free only for A — it must also be free for B, since B will inherit alpha. As will be seen, this correction to the André-Royer algorithm implies that the code for the separate and declared partitions are identical and can be merged.

5. Supposing that selector S is not free for C (and its subclasses) as discussed in Note 4, André-Royer states that a new color can be obtained by finding a color free for all classes using the current color. This is only true if we assume that C and its subclasses are considered to be using S at the time that S is being defined in C. To

make this assumption clear, the test for the color free for class C and all of its subclasses is made explicit.

6. If a selector S is not defined in the completeBehavior of any of the immediate superclasses of C but is defined in the subBehavior of C, then partition(S) is *declared.*

7. The test as proposed by André-Royer is inefficient. The second test can be avoided entirely, and the number of subclasses tested reduced by asking only for dependent subclasses (those subclasses which inherit S from C).

8. As in note 4, we must also insure the color is free for the dependent classes of C. Note that here, the number of dependentClasses(C) will generally be substantially less than the number of allSubClasses(C) since at least one subclass re-defines S (otherwise, the partition would be separate instead of declared). Since all classes in the set must be checked, it is desirable to have as small a set as possible for efficiency.

9. If a selector changes color, the divisions for all classes within the cache-table that use that selector must be moved from the old color row to the new color row.

```
Algorithm MAR(in C : Class, in/out S : Selector, in/out T: CacheTable)

    let L = color(S)

    Switch partition of S
        case specific:
            find free color for allSubClasses(C) ∪ {C}
        case redefined:
            no change
        case separate:
        case declared:
            If for $C_*$ ∈ dependentClasses(C),
              ( currentColor(S) ∈ freeColorsFor($C_*$) )
            no change
            else
            find color free for classesUsing(currentColor(S)) ∪ dependentClasses(C)

    If L ≠ color(S)
        for $C_i$ ∈ classesUsingColor(L)
            divisionAt[color(S), $C_i$] = divisionAt[L, $C_i$]
            divisionAt[L, $C_i$] = Ω
```

Figure 12 : Modified André-Royer Algorithm.

4.3. The ICP Algorithm

The ICP algorithm is an extension of the modified André-Royer Algorithm presented in Figure 12. As discussed previously, the most substantial difference between the André-Royer and ICP algorithms is the importance attached to partitions and the efficiency with which partition types can be determined for selectors. In order to provide the most efficient selector-to-partition determination, the ICP algorithm incrementally maintains partition information in addition to color information.

4.3.1. ICP Dispatch Classes

Four classes are defined in the PE to implement the ICP algorithm. In the discussion of the André-Royer algorithm, there was no mention of how a color was associated with a selector, and a variety of other details were ignored. Here, we present the classes used to store all information needed for ICP.

4.3.1.1. MSSelector

An MSSelector is an encapsulation of the information associated with a MS selector, and consists of a unique *index* (Integer), *color* (Integer), *name* (Symbol) and *msClasses* (a Set of defining classes).

When a method declaration is parsed, the MS message selector name (an ST-80 Symbol) must be registered by sending a message (*addSelector:toClass:method:behavior:*) to the appropriate cache table (instance or class). A check is made to see if an instance of MSSelector exists for the selector (i.e. an MSSelector instance whose *name* state matches the selector). Each cachetable maintains a *selectorMap* which provides an efficient method of mapping between ST80 symbols (representing MS selectors) and MSSelector instances (which encapsulate all information associated with a MS selector). If no such MSSelector instance is found, one is created with a new index. The *name* is set to the ST80 symbol, and *msClasses* is initialized to a Set with one element — the instance of MSClass currently being parsing. The state *msClasses* stores only the instances of MSCClass which explicitly define the selector. Classes which respond to the selector due to inheritance are *not* stored. Finally, *color* is set by executing the ICP algorithm as discussed below. This implementation provides the best trade-off between execution efficiency and space usage.

4.3.1.2. MSDivision

During the discussion of the André-Royer it was mentioned that cache-table entries must store both a method and a selector. Within ICP, a substantial amount of additional information is needed at compile-time, although no additional information is needed at run-time. Although the André-Royer revolved around a discussion of selectors, the ICP algorithm places more emphasis on divisions (cache-table entries). A single division is used to represent a group of classes which use the same method for a specific selector. For each selector, ICP logically divides the classes that recognize it into mutually exclusive MSDivisions. Each division determines a set of classes that form a connected component of the inheritance hierarchy and use the same method for a given selector. The root of this sub-graph is the *defining class* for the selector (division) and the additional classes in the connected sub-graph are called *dependent classes* because they are dependent on the defining class for the implementation of the method associated with the selector. A *non-dependent class* is a subclass of the defining class or any dependent class of the defining class which redefines the selector. A *non-determined dependent class* is a dependent class which has non-dependent subclasses, and a *determined dependent class* is one which does not have any non-dependent subclasses.

The state of a division includes a selector, a defining class, a method, a set of non-dependent classes, a set of non-determined dependent classes, and a flag denoting whether superclasses define the associated selector. The information stored in a division provides the optimal combination of space efficiency and partition-type determination. Note that since every division has a selector, a property of a selector can also be considered a property of a division. In particular, in the discussion below, we will talk of divisions having a particular partition type, and this is to be understood to mean that the selector associated with the division has the given partition type. Note that the information stored in each division allows for an immediate determination of its partition type without any tree searching required.

MSEmptyDivisions are pecial subclasses of MSDivision used when a new row or column is first added to a cache-table. By treating empty divisions in a manner identical to full divisions, no special case code needs to be written.

4.3.1.3. MSCacheTable

An MSCacheTable contains a two dimensional array (matrix) with rows indexed by colors and columns indexed by class indexes. The values of the matrix are MSDivisions. An MSDivision represents a set of classes which use a common method for a selector. The selector specifies a color, and the set of classes specify a set of class indices. Within the matrix, a row is specified by the color. The set of columns (classes) from this row identified by the MSDivision all contain the same MSDivision object. Any class that does not support a selector for a given color contains an instance of MSEmptyDivision in its column. The MSCacheTable supplies other services in addition to the cache matrix.

As an example, suppose a hierarchy exist such that B and C inherit from A, which is the root of a tree, and E and F inherit from D, also the root of a tree. Classes O and G are defined but have no behavior. Both A and D separately define a selector *alpha* (with color 1), and it is not redefined any any of the other classes. Suppose O has class index 1, and classes A-G have indicies 2-8 respectively. One MSDivision instance is used to represent A, B and C; columns 2, 3 and 4 of row 1 of the cache table will store this MSDivision object. A different MSDivision instance represents D, E and F; columns 5, 6 and 7 all store this object. Table 6 shows the resulting cache table.

Color	O	A	B	C	D	E	F	G
1	EmpDiv1	Div1	Div1	Div1	Div2	Div2	Div2	EmpDiv2

Table 6 : Example MSCacheTable

In Table 6, EmpDiv1 and EmpDiv2 are instances of MSEmptyDivision, and Div1 and Div2 are instances of MSDivision.

At run-time, when executing a message send, the MSMessageSend node knows its receiver and MSMessage. The receiver knows its class index. The MSMessage knows its selector index. To find the method, the selector index is used to obtain a MSSelector instance, which in turn is used to obtain a color index. The color index and class index are used to access a color/class entry in the cache. Note that a run-time check must be performed to ensure that the selector associated with the stored method is the same as the current selector (due to the coloring algorithm mapping multiple selectors to the same color). This overhead will be discussed in more detail later. The current implementation contains two cache tables, one for instance methods and one for class methods. For each selector, the compiler generates two indices, one for each cache. At run-time, the appropriate index is used, depending on whether the receiver is a class object or not. It is possible to combine these two cache tables into one, but the compiler must still generate two indices because the same message selector may be used for a class and an instance message and there is no way to tell at compile-time whether the receiver is a class or not. Additional research on the merits of one cache table versus two as well as multiple cache tables, one for each tree in the inheritance forest, is needed.

4.3.1.4. MSConflictSet

An MSConflictSet is responsible for resolving message selector name conflicts caused by multiple inheritance. An instance of MSConflictSet is used for every class parsed. During parsing, after obtaining the superclasses for the class, the conflict set is initialized with all inherited behavior. Using a double dictionary, with selectors as primary keys, MSDivisions as secondary keys, and classes as values, the MSConflictSet records conflicts. A conflict exists for any primary key which has more than one secondary key. Remember that a MSDivision represents one particular method implementation, so different MSDivisions represent different implementations. During the parsing of the native behavior, such conflicts can be removed in one of two ways. First, if a selector is defined in the native behavior, all secondary keys for that primary selector key are removed, since native behavior has precedence over inherited behavior. Second, an alias can move one of the secondary MSDivision keys to a different primary key.

The following is an example to show the usage of the MSConflictSet. Suppose C inherits from A and B, both of which define different versions of *alpha*, *beta* and *gamma*. Only B defines *delta*. C redefines *gamma* and aliases the *beta* inherited by B to *epsilon*. During the parsing of the class declaration of C, the superclasses are obtained. At this point, a new MSConflictSet instance is made and initialized.

Primary Key	Secondary Key	Value
alpha	Div1	{A}
	Div2	{B}
beta	Div3	{A}
	Div4	{B}
gamma	Div5	{A}
	Div6	{B}
delta	Div7	{B}

Table 7 : MSConflictSet before parsing native behavior

Currently conflicts exist for *alpha*, *beta* and *gamma*. Upon parsing the native behavior, the conflict set looks like this:

Primary Key	Secondary Key	Value
alpha	Div1	{A}
	Div2	{B}
beta	Div3	{A}
gamma	Div8	{C}
delta	Div7	{B}
epsilon	Div4	{B}

Table 8 : MSConflictSet after native behavior

Notice that the conflict for *beta* no longer exists because Div4 has been moved to selector *epsilon*. The conflict for *gamma* was removed because C explicitly defined a *gamma*. However, a conflict still exists for *alpha*, so a parsing error will be generated stating where the conflict exists. It is at this time that the value of the double dictionary is used — the class(es) stored are printed out in the error message.

4.3.2. The Incremental Coloring Algorithm (ICP)

As mentioned above during the discussion of André-Royer, the part of the algorithm associated with class declarations remains the same. It is the algorithm for method declaration that is substantially changed.

The definitions in Table 9 below are needed in addition to the definitions presented in the André-Royer section.

Symbol	**Definition**
divisionDefiningClass(D)	the root class of the division D
superDefined(D)	boolean stored value of a division denoting whether its associated selector is defined in the superBehavior of its associated class.
nonDependents(D)	$= \{C_i \mid C_i < C$ and $S \in$ definedBehavior(C_i) and if $S \in$ definedBehavior(C_j) then $C_j < C_i\}$
nonDetDeps(D)	$= \{C_i \mid C_i \in$ dependentClasses(D) and divisionSelector(D) $\in$ subBehavior(C_i)
superClassChain(C_s,C)	$=\{C_i \mid C_i < C$ and $C_s < C_i\}$

Table 9 : Definitions for the ICP Algorithm

Table 10 defines partition types using the stored division information, and these definitions can be seen to be much more efficient than André-Royer. In the table, D represents a division and C is the divisionDefiningClass(D).

Partition	**Definition**
specific(D)	= [classesDefiningSelector(S) = $\{C\}$]
separate(D)	= [not superDefined(D) and nonDependents(D) = Ω]
internal(D)	= [superDefined(S) = true and nonDependents(D) $\neq \Omega$]
determined(D)	= [superDefined(S) = true and nonDependents(D) = Ω]
declared(D)	= [not superDefined(S) and nonDependents(D) $\neq \Omega$]

Table 10 : Partition types for divisions

4.3.3. The ICP Algorithm

Algorithm ICP is called when a selector S is added to a class C.

```
Algorithm ICP(in C : Class, in S : Selector, in/out T: CacheTable)                         NOTE 1

    let D = a new division
    let L = color(S)

    let divisionSelector(D) = S
    let divisionDefiningClass(D) = C

    If specific(D) then                                                                    NOTE 2
        set color(S) from colorsFreeFor(dependentClasses(D))
    else
        let E = divisionAt[C, L]
        if (divisionDefiningClass(E) ≠ C) then                                             NOTE 3
            if ( ( divisionSelector(E) ≠ Ω) and (divisionSelector(E) ≠ S) ) then           NOTE 4
                MoveSelectorToFreeColor(divisionSelector(E))
            endif
            if ( superDefined(E) = false ) then                                            NOTE 5
                if (dependentClasses(D) ∩ classesUsingColor(L) = ϕ) then
                    no color change.
                else
                    let G = classesUsingColor(currentColor(S)) ∪ dependentClasses(D)
                    set color(S) from colorsFreeFor(G)
                endif
            else
                no color change
            endif

            for each C in nonDependentClasses(D)                                           NOTE 6
                divisionPartition(D) += superClassesChain( C , C )

            for each C in superclasses(divisionDefiningClass(D))                           NOTE 7
                D1 = divisionAt( C , currentColor(S) )
                nonDependents( D1 ) -= nonDependents( D ) union {C}
                divisionPartition( D1 ) -= divisionPartition( div )

        endif                                                                              NOTE 8
    endif

    for C_i in dependentClasses(D) ∪ {D}                                                   NOTE 9
        set divisionAt[C, color(S)] = D
    endfor
end ICP
```

Figure 13 : The ICP algorithm

Notes

1. Before selectors are added to a class, the ICPNewClass Algorithm must be called to insure that a column exists in the cache-table for the given class C and that the initial divisions stored within this column are correct.

2. Partition is specific — use modified André-Royer for specific partitions. Namely,

remember that the new color must be free for C and its subclasses. Checking just the subclasses is not sufficient, since inheritance exceptions do exist in MS. For specific paritions, dependentClasses(D) is the same as C and all of its subclasses, if definingClass(D) = C.

3. If the defining class of the division equals the current class, a selector in C is being recompiled — it has already been defined in C. Thus, absolutely nothing needs to be done, so there is no 'else' clause to this primary 'if' condition.

4. Even though the division is obtained by accessing the cache-table at the specified class index and selector color, there is no guarantee that the division obtained is for the selector in question (since there are multiple selectors per color). If the division accessed does not reference the current selector S, one of the two selectors must be moved to a different color — the old selector is chosen arbitrarily to be moved. If selectors stored all classes that use them, the selector using the fewer classes could be moved to improve time efficiency, but the amount of space needed to store this information is prohibitive. Of course, if it was determined that the new selector should find another color instead of moving the other one, this is done simply by newColor = colorsFreeFor(classesUsingSelector(S) $\cup$ allSubClassesAndSelf(C)). The determination of which is more efficient, moving the old selector or obtaining this new color, is non-trivial. Since the relative improvement in efficiency will be minimal, the issue is not of great concern.

5. If superDefined(E) = false, this implies that no superclass defines selector S, so the partition is declared or separate — use the modified André-Royer algorithm for separate or declared partions.

6. Since the set of nondependents for a division specify those classes which have the associated selector S defined, all classes between these classes and the defining class C must be added to the division partition — the set of dependent classes which have definitions for S in their subBehaviour.

7. Since the set of nonDependents includes only the closest subclasses redefining S, if a new division is added, all nonDependents of that division (and C, the defining class of that division) are no longer recorded in the nonDependent information of superDef classes. Similiarily, since the partition represents only dependent classes with certain properties, adding a new division makes any classes in the partition of this new division not dependents of any superDef divisions.

8. No modification need to be made to the dependents of the division, since this information is not stored behavior, due to space considerations. The number of nonDependent classes will in general be much smaller than the number of classes inheriting any specific behavior. Since dependent information is not needed at the time of repartitioning, it is not stored. The only time it comes into play is when determining the partition of selectors in classes, as discussed below.

   Furthermore, no modification is made to the superDefs of the division since this information is currently not stored behavior of a division. One superDef is always trivial — by looking at the current definingClass of the division. However, more superDefs may exist due to MS's alias capability — some superClass may have defined the selector and an intermediate superclass aliased the selector away (so, from the perspective of the selector, it has been deleted). Currently, it is easier to simply search up the tree when superDef information is needed than it is to incrementally modify this information if it were stored. More research into relative advantages and disadvantages is needed.

   Note that a boolean flag determining whether there are any redefinitions is stored within each division. This flag is necessary to allow divisions to determine the partition of its selector immediately.

9. After initializing the new division D, a reference to it is stored for every dependent class of it.

4.3.3.1. An ICP example

Figure 14 displays 6 inheritance graphs representing differing stages of selector definition. The first graph has no behavior defined for any of its classes, the second has selector *b* defined for class B, and so on. Below a step by step execution of the ICP algorithm using this figure. The example has been designed to demonstrate all of the recoloring code within the algorithm. Due to the complexity involved in the full algorithm, the reader is referred to [HS93] for a complete discussion of both recoloring and repartitioning.

When defining *b* within B the ICP algorithm indicates that a new division should be made, whose divisionSelector is *b* and whose definingClass is B. The selector and defining class uniquely identify a division, thus in the example below divisions will be denoted by *selector:className*. For example, the newly created division is denoted b:B.

Since the selector *b* is unique in the environment, it is specific. According to ICP, this implies that the associated color of *b* can be any color free for the dependentClasses(b:B). Since dependentClasses(b:B) = { B }, and no colors currently exist in the environment, color 1 is free and can be used.

After setting the color of *b* to 1, the algorithm specifies that for C ∈ dependentClasses(D), the cache-table entry at color 1 and class C should store the division b:B.

Color	A	B	C	D
1	Ω	b:B	Ω	Ω

Table 11 : CacheTable after b:B

When defining *c* within D, the same code is used as for *b* above. Since dependentClasses(c:D) = {D} and color 1 is free for D, it can be used to store *c* as well as *b*.

Color	A	B	C	D
1	Ω	b:B	Ω	c:D

Table 12 : CacheTable after c:D

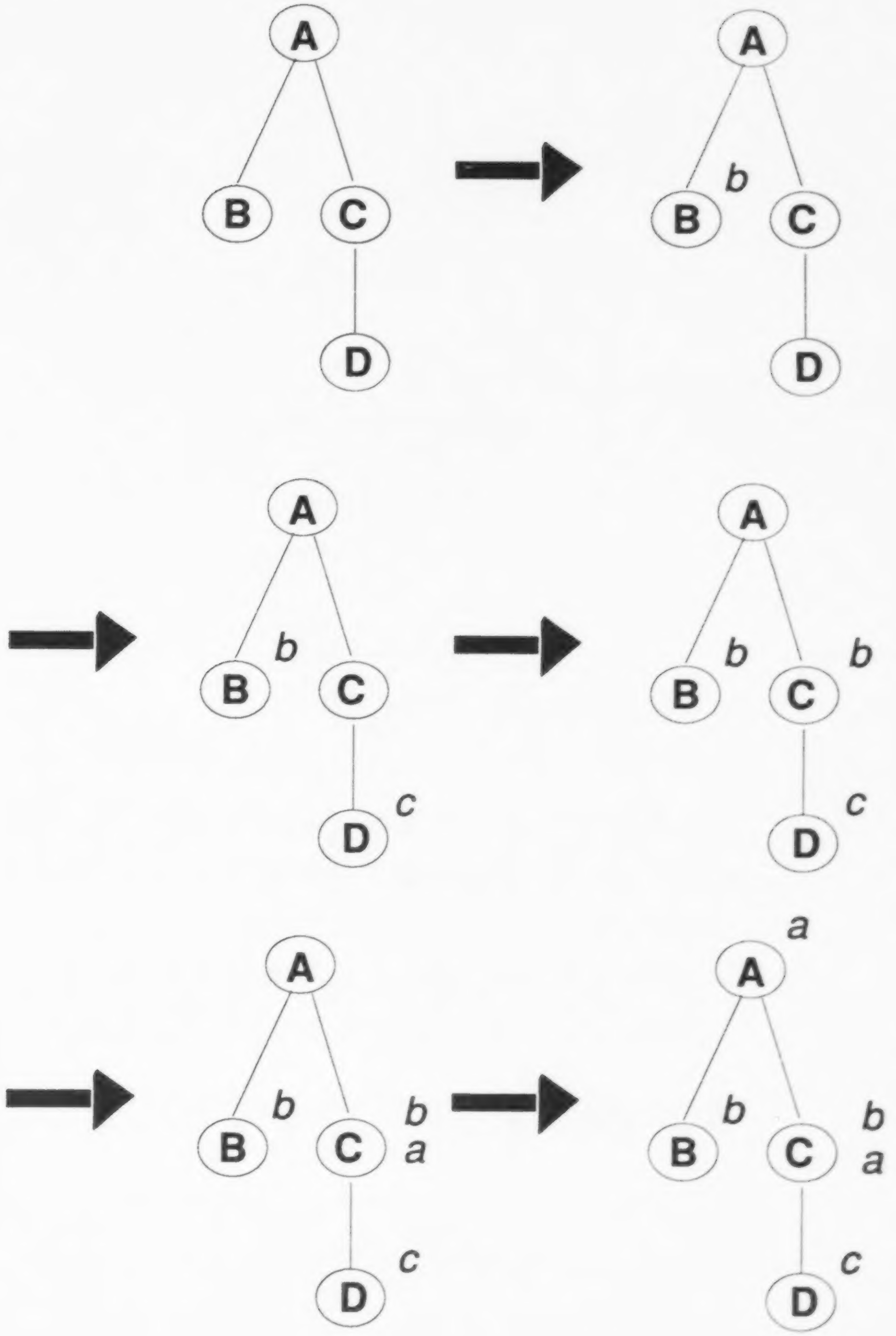
A
B
C
D
A
b
B
C
D
A
b
B
C
c
D
A
b
B
b
C
c
D
A
b
B
b
C
a
c
D
a
A
b
B
b
C
a
c
D

Figure 14 : Example ICP Inheritance Graphs

When defining *b* in C, new code within the ICP algorithm is encountered. Division b:C is created by making an new instance of MSDivision and associating selector *b* and definingClass C with it. Since the division is not specific, ICP states that the current division stored at class C and the color associated with *b* (namely color 1) should be obtained. From Table 12 it can be seen that E = Ω. Since E is an empty division (denoted here by Ω), its divisionDefiningClass is not equal to C. The algorithm skips the MoveSelector procedure call because E is empty. Since superDefined(Ω) is false, a check is made to see if the intersection of dependentClasses(b:C) and classesUsingColor(1) is empty. The first set is {C, D}, the second set is {B, D} and the intersection is {D}, which isn't empty. Hence, let G = classesUsingColor(1) ∪ dependentClasses(b:C) = { B, C, D}. Choose any color from colorsFreeFor(G) = { 2, 3, ... }. Set color(S) = 2. Next, move the divisions associated with all classes defining S from the old color to the new color. Thus, division b:B is moved from row 1 to row 2. After the partition updating information, the final part of the algorithm stores the new division D into the appropriate cache-table locations, namely row 2, in columns for class C and class D.

Color	A	B	C	D
1	Ω	Ω	Ω	c:D
2	Ω	b:B	b:C	b:C

Table 13 : CacheTable after b:C

When defining *a* in C, partition type is specific and dependentClasses(Div) = {C, D}. Note that colorsFreeFor(dependentClasses(Div)) = { 3, 4, ... }, so since color(S) = 1 is not in this set, we pick a new color, say color(S) = 3. The division is stored into C and D as required.

Color	A	B	C	D
1	Ω	b:B	Ω	c:D
2	Ω	b:B	b:C	b:C
3	Ω	Ω	a:C	a:C

Table 14 : CacheTable after a:C

Finally, when defining *a* in A, E = Ω, superDefined(E) is false, dependentClasses(D) ∩ classesUsingColor(3) is {C, D}, G = { A, B, C, D }, colorsFreeFor(G) = { 3, 4, ... } so we keep color(S) = S. The dependent classes of a:A are {B}.

Color	A	B	C	D
1	Ω	Ω	Ω	c:D
2	Ω	b:B	b:C	b:C
3	a:A	a:A	a:C	a:C

Table 15 : CacheTable after a:A

In this example, a in A is partition type declared, a in C is partition type determined, b in B is separate, b in C is separate, and c in D is specific. If *a* were defined in D, *a* in C would become internal, and *a* in D would be determined.

For a complete discussion of the algorithm, including the incremental partitioning information, the interested reader is referred to [HS93].

4.3.3.2. Using Partition Types to Avoid Method Dispatch

Within the ICP algorithm, the *redefined* André-Royer partition type has been separated into two partitions. For André-Royer, the definition was : S is redefined if S is in completeBehaviour(C). In MS, though, for a class C which redefines S, a distinction is made between whether the selector S is redefined further in subBehaviour or not. The importance of this distinction will be seen later. Thus, instead of redefined, the partition types *internal* and *determined* have been defined in Table 10.

There are a variety of times in which no run-time lookup is needed. If the receiver of the message is self or a literal, the class is known, and thus the method associated with any selector is known. Furthermore, if any selector has partition type *specific*, no lookup is needed, since this partition type implies that there is only one class implementing the selector.

The true use for partitions, however, comes with static typing. If MS were extended to provide static typing, more runtime lookup can be avoided. Suppose each variable, argument and message return specifies at least the root class of the tree of the possible classes of the expected object values. Then selectors with partition type *separate* will also be uniquely defined. If variables and methods are typed more finely, to some set of specific class types, then if, during a message send, the selector for all of these classes has partition type 'determined', no runtime lookup is required.

Given this, the rationale behind the design of the MSDivision object becomes apparent; given a MSDivision, the partition type of its associated selector can be determined immediately. If the selector has only one defining class, it has partition type *specific* (Note that determining if a selector is specific does not rely on the MSDivision at all). Otherwise, if superDefined(D) is true, the selector is partition type *internal* or *determined*, depending on whether nonDependents(D) is non-empty or empty respectively. Finally, if superDefined(D) is false, the selector is partition type *declared* or *separate*, depending on whether nonDependents(D) is non-empty or empty.

Figure 15 shows a small inheritance graph based on the graph from [AR92], but with an inheritance exception added. The exception occurs by renaming the *o* message to *a* in class A (denoted o -> a). Table 16 shows the results of coloring this graph using the ICP algorithm.

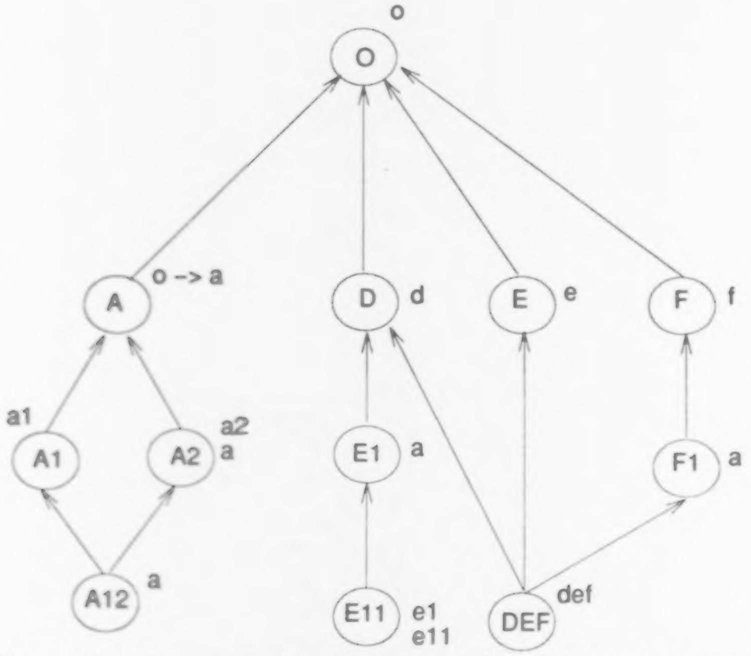

Figure 15 : An example inheritance graph with an inheritance exception

L/C	O	A	A1	A2	A12	D	E1	E11	E	F	F1	DEF
1	Ω	a:A	a:A	a:A2	a:A12	Ω	a:E1	a:E1	Ω	Ω	a:F1	a:F1
2	Ω	Ω	a1:A1	Ω	a1:A1	Ω	Ω	Ω	Ω	f:F	f:F	f:F
3	o:O	Ω	Ω	a2:A2	a2:A2	o:O	o:O	o:O	o:O	o:O	o:O	o:O
4	Ω	Ω	Ω	Ω	Ω	Ω	Ω	e1:E11	e:E	Ω	Ω	e:E
5	Ω	Ω	Ω	Ω	Ω	Ω	Ω	e11:E11	Ω	Ω	Ω	def:DEF
6	Ω	Ω	Ω	Ω	Ω	d:D	d:D	d:D	Ω	Ω	Ω	d:D

Table 16 : An ICP generated coloring for the inheritance graph of Figure 15

To test the algorithm on real graphs, we developed an ST-80 to MS conversion program. We are translating the Collection classes and modifying them to reflect the multiple inheritance class re-factorization developed by [Coo92]. So far, we have translated and re-factored the ST-80 collection classes: Collection, SequenceableCollection and Set to the MS classes: Collection, IndexedCollection, SequenceableCollection and Set. The ICP algorithm generated 49 colors for the 90 selectors.

4.4. Space Efficiency (Tail Removal)

There are several ways to reduce the run-time space requirements for cache table dispatch. These reductions are often obtained by increasing compile-time space requirements which in most cases is not disadvantageous. Suppose the basic cache-table has size N (colors) x M (classes). Let n be the color index of the last non-empty selector in a class column, m. The entries from row index n+1 to N are called the tail of column m and this tail can be discarded. Therefore the cache-table can be implemented as an array of columns of variable size.

Note that the specific color associated with a group of selectors is not important; only that the selectors in the group have the same color. Thus, entire rows of the cache-table can be swapped. To optimize the size of the cache table, rows should be swapped to maximize the sum of the tail sizes.

4.5. The Method Dispatch Algorithms

Appendix B contains the code executed at run-time for the ICP method dispatch algorithm and for the ST80 lookup dispatch algorithm. The information required (at run-time) to perform method dispatch in each case is listed in Table 17

	Cache Table Approach	Class Look-up Approach
Divisions	a selector and a method	a method and a visibility (public/private)
Selectors	a unique index and a color index	a unique index
Classes	an index	an index and a dictionary of divisions

Table 17 : Method dispatch information

The current implementation uses the table approach with separate cache-tables for class and instance selectors with tails removed. For table dispatch, the appropriate selector index and cache-table are determined at run-time by determining whether the receiver is a class or instance object. Next, the color/class entry in the appropriate table is obtained. This entry is a division, but its method is not necessarily the correct one. Two exceptional conditions must be checked for; the division may be empty or the division may define a selector that is different than the one in question. In either case, the desired selector is not understood by the receiver, and an appropriate error message is generated. If neither of these cases occur, the division specifies the method to execute.

For class look-up dispatch, the execute algorithm simply asks for a division by calling lookup_dispatch. If the resulting division is empty, a messageNotUnderstood message is sent, otherwise, the associated method is executed and returned as the result.

The lookup_dispatch algorithm is recursive. It needs a receiver, a current class, and the instance and class selectors. The current class is initially the class of the receiver, but may change in recursive calls. The behavior type of the receiver (instance or class object) is used to obtain the appropriate selector index, as well the proper method dictionary from the current class. Next, the specified selector is searched for in the method dictionary. Even if the selector exists, a test must be made to determine whether the method is private. If it is, the method is only applicable if the class of the receiver is equal to the current class (definition of private in MS). If a method is found and is legal, its division is returned. Otherwise, a recursive call must be made to lookup_dispatch, with the receiver and selectors remaining the same, but with the current class being changed to one of the superclasses of the current class. If the result of this recursive call is an empty division, it is attempted again on another superclass, until a non-empty division is found, or all super classes have been searched. The resulting division is returned as the result of the algorithm, even if it is empty.

Table 18 contains comparative performance results for the class look-up and coloring dispatch algorithms, based on illustrative code that will not occur in practice. Subclasses of the array class were created at depths of: 1, 5, 10, 15 and 19. Note that although a tree depth of 19 is unrealistic, a class having four superclasses which are each 5 classes removed from a root is not nearly as unrealistic, and from the perspective of look-up dispatch is the same as a depth of 20.

The same array initialization message was sent to instances of the leaf node classes of these inheritance chains and the time was recorded. The array initialization code is given in

Appendix A for the subclass at depth 1. In every case, an array size of 50 and iteration size of 5000 were used so that a total of 250,000 dispatches were done. The trials were done on a SPARC ELC and the average of 10 trials was used in each case.

Depth			
1	19,400	24,500	26%
5	10,400	24,500	135%
10	5,720	24,500	328%
15	3,750	24,500	553%
19	2,980	24,500	722%

Table 18 : Method dispatch comparison

Chapter 5 : C-Code Generation

As discussed in Chapter 4 the object-oriented approach to parsing is substantially different than the convential approach. Here, the node classes representing synactic and executable fragments of MS programs each have behavior to parse themselves, executes themselves and generate C-code for themselves. This chapter describes the C data-structures used and explains how the parse nodes within the PE generate stand-alone C code.

The list of goals described in Chapter 4 is applicable to the C code as well as the PE. The primary goal is efficiency, but implementation flexibility is also important. In order to provide for such flexibility without reducing efficiency, the code generated by node classes consists entirely of macros. Macros provide a level of abstraction, allowing different implementations of the macros to exist so as to determine the effects of such implementations. Furthermore, the reliance on macros implies that the code generated by the PE is not actually C-code. Since the macros discussed below are defined in C-code, we have been refering to the C-code implementation. However, the macros can just as well be defined to expand into assembler language or any other language of choice. Finally, using macros makes the generated code much more readable.

Within the current implementation there are two levels of macro definitions: *cgen-macros* and *access-macros*. The node classes generate text representing *cgen-macros* which expand out into artibrarily complex C code. However, because implementation flexibility is a priority, some mechanism for testing differing implementations of various data-structures is desirable. To provide such a mechanism, access to all data-structures (both retrieval and storage) is obtained by using *access-macros*. Thus, in the C-code written to define cgen-macros, access-macros are used whenever information is required from a data-structure or needs to be added to a data-structure. For example, there are many access-macros for accessing/modifying the C language data-structure representing an MSObject, such as OBJ_CLASS(obj) or OBJ_NAMED_AT(obj,i). The first one returns the class object of the object *obj*, and the second one returns the *i*th named instance variable within object *obj*.

Before discussing the code generated by individual node classes within the PE, the data-structures and auxillary code needed to implement an object-oriented language in C are described.

5.1. C-Structures

Five primary data-structures exist in the C-code implementation: MSObject, MSPEnv, MSModule, CacheTable, and MSContext.

5.1.1. MSObject

The most important C data structure, called an MSObject, is used to represent an MS object. For efficiency reasons, objects are dynamically allocated arrays. In this section, stored behaviors will be referred to as instance variables, where each instance variable is named, indexed or byte-indexed. Each object also stores a flag denoting whether the object is the result of an explicit return from a method, a flag denoting whether the object is immutable or not, the position within the object of the first indexed instance variable and first byte indexed instance variable, and the size of the object. Indexed and byte indexed instance variables are implemented as arrays, whose first element is the MS integer object denoting the size of the variable, and whose successive elements are the values of the variable.

Type	**Specification**	**Description**
word 0	class_ptr	Address of object representing the class of this object.
word 1 : Bit 0	return_flag	If 1, this is a return object.
word 1 : Bit 1	literal_flag	If 1, this is a literal object, and is immutable
word 1 : Bit 8-15	obj_size	Size in words of this object (maximum is 256)
word 1 : Bit 16-23	first_idx	Index within MSObject array of first byte indexed variable
word 1 : Bit 24-31	first_byteIdx	Index within MSObject array of first indexed variable
Word 2	state	Start of state information.

Table 19 : MSObject array

5.1.2. Classes as MSObjects

For uniformity, MS class objects are stored as MSObjects. Since class objects require space to store a variety of information not needed to be stored by instance objects, this space is added after the first two words and before the start of the class named instance variable state, and is considered *hidden* state. Such information includes the number of named, indexed and byte indexed variables for instances and classes, and the unique integer index representing the class.

5.1.3. MSPEnv

The entire execution environment of a program is stored in an MSPEnv structure. Each program has one global variable of this type, accessible within any file in the program. Besides fields to provide for memory allocation and diagnostic printing, the MSPEnv structure has an array of modules, a context stack, a list of all literals statically specified in the program, and cache tables for instance and class selectors.

Type	Specification	Description
MSObject *	class[MAXIMUM_CLASS_NUMBER]	Only needed for st80_dispatch
MSModule	module[MAXIMUM_MODULE_NUMBER]	All modules in program
MSObject *	context[MAXIMUM_CONTEXT_DEPTH]	Stack representation of dynamic link
MSObject *	literalContext	used by literal blocks
MSObject **	literals	array of literal MSObjects specified statically within program
CacheTable	instTable	Table for methods
CacheTable	classTable	Table for class methods
MSObject *	hardcode[NUM_HARDCODE]	
int *	InstSelectorColor	Array mapping instance selector index to color
int *	ClassSelectorColor	Array mapping class selector index to color
char **	ClassName	Array mapping class index to class name - diagnostics only
Memory *	G_memory[MEMORY_MAX]	memory for program - garbage collection facility

Table 20 : MSPEnv structure

5.1.4. MSModule

A module is represented by an MSModule structure that contains a context and an integer representing the index of the module within the environment's module array.

Type	Specification	Description
MSObject *	context	instance of class MSContext
int	moduleIndex	index within MSPEnv module array of this module

Table 21 : MSModule structure

5.1.5. CacheTable

The entries in the cache tables store three values : a C function address, the index of the selector associated with the C function, and a C string representing the function name. The C string is needed only for diagnostic purposes and can be omitted for space optimization (see Section 4.4). In addition, since the tail-removal space optimization technique has been implemented, the cachetable stores information associated with each column, namely the index of the last non-empty cache table entry within the column. If a particular color/class entry does not have a value, the function address index and function name index are NULL, and the selector index is -1.

Type	Specification	Description
int	numColors	
CacheRow *	colors	Array of substructures representing a single row (color)

Table 22 : CacheTable structure

Type	Specification	Description
int	numClass	size of the three parallel arrays belows
MSFuncPtr *	functions	Array of function addresses
int *	selectorIndex	Array specifying selector index at a given class index
char **	functionName	Array specifying function name at given class index (diagnostics)

Table 23 : CacheRow structure

5.1.6. MSContext

There are two useful ways in which contexts can be implemented. From a object-oriented perspective, the best implementation is to define a Context class within MS, and have contexts be instances of this class (the meta-object approach). This would allow users to manipulate the contexts during execution — a useful feature during debugging. On the other hand, from an efficiency perspective, a simple array representing the context is desirable (dedicated-structure approach). To study the relative advantages of both implementations, access to contexts are provided by access-macros, and both implementations are provided.

Any implementation of contexts must provide a facility for recording the static link of the current context, where the static link is itself a context. Since contexts exist only for modules and blocks, and blocks are always associated with a module, every block, no matter how deeply nested, has the concept of its associated module context. Finally, contexts must also record the size of the context they are storing, as well as providing space and efficient accessing mechanisms (i.e. an array) for elements. Each element is a MSObject, and the index of the array is the offset of that element within the context.

The issue of meta-object representation of implementation code requires more study to determine its advantages and disadvantages. It is possible to implement the MSPEnv structure as a MSObject, thus allowing programs to view the entire run-time environment, but this requires that every component of the MSPEnv structure also be an MSObject. Preliminary analysis does not reveal any unsurmountable problems with this approach, and thus future versions of the implementation can be expected to provide this MSObject representation of all primary C structures.

A *static context chain* consists of a context and its recursive static links. To obtain a particular object from a static context chain, a level and an offset are required. The level is an integer relative to the current context specifying how many times to follow the static link. For a level of 0, this implies that the context to use is the current one. For a level of n, the static link of the current context is followed 'n' times, and the resulting context is used. Once the appropriate context has been obtained, the 'offset' is used to obtain the correct object.

Each context is part of a static context chain specifying the nesting of the associated blocks. The end of all static context chains is a module context (module contexts do not have a static link). In addition to this static context chain, there is a *dynamic context chain*, representing what block is currently being executed, and thus, representing the *current context*. When a block is executed, its context is added to the front of this dynamic chain, and when the block has finished execution, the context is removed from the dynamic context chain. Within the current implementation, the dynamic context chain is represented as a stack in MSPEnv.

5.1.7. State

Although indexed and byte indexed state can be implemented in a variety of ways, the most efficient is a simply array whose first index stores an MSObject of class Integer representing the size, and whose succesive indicies store the corresponding values of the indexed state. Note that the size is stored as an MSObject of class integer for efficiency reasons — MS programs can ask for this size, and such a request must return an MSObject. By storing the object explicitly, no run-time inefficiency is introduced creating one.

5.2. Primitive C Functions

MS methods are mapped to C functions in the C-code implementation. Generated functions for MS methods have a common argument list which is also shared by all primitive C functions. Thus, the first argument is the receiver *rec* of type MSObject *, the second argument is an array of arguments *args*, where each argument is an MSObject * type, and thus the array is an MSObject ** type. Finally, the third argument is an integer *size* representing the size of the argument array. The only constraints put on primitive functions is that they return MSObject * results. Primitives should always use macros to access data-structures. Furthermore, many macros have been defined to make primitive writing easier.

5.2.1. undefinedMethod and abstractMethod

Modular Smalltalk has the concept of *undefined* and *abstract* methods, both of which do not have user specified executable code associated with them. Any method defined as such will end up calling the primitive *undefinedMethod* or *abstractMethod* respectively. The language specification does not give details on how to handle such circumstances. The current versions print out a diagnostic error explaining that execution of an undefined or abstract method was attempted, and stops execution.

5.2.2. msBasicNew

The *msBasicNew* primitive function has a class object as an argument and is used to create a new instance object of that class, and to initialize the state of this new object. Named instance variables are initialized to nil and the size of indexed instance variables is set to zero.

5.2.3. Required Class primitives

A variety of primitives must be implemented for the required classes in order to give the language some functionality. These include the *value* and *value:* messages for Closures, accessing methods for arrays, arithmetic operations for integers and floats, etc.

5.3. Library Functions

In addition to the C generated code from the ST-80 PE and any C primitives needed, there is surprisingly little additional code needed in order to provide for an executable MS program. The following sections describe the library functions required.

5.3.1. execute

One C function is responsible for implementing the method dispatch at run-time. Each time a message-send occurs, *execute* is called with arguments specifying a receiver, message arguments, class index, instance selector index and class selector index. The receiver is used to determine if it is a instance or class object, and the appropriate color for the corresponding selector index is obtained and used along with the class index to obtain an index into the proper cachetable. Appendix B presents this function and provides a small discussion about it.

5.3.2. makeClass

The *makeClass* function is used to create a new class object. Its arguments consist of : an instance and class dictionary of methods and associated visibilities (needed only if a ST80-like dispatch mechanism is used), a set of superclass indices, integers for number of variables (of all types), and finally, an integer class index. These arguments are used to initialize the default elements of the newly created class object.

This is not a primitive function because the language specification states that class descriptions are not objects, and thus new classes cannot be made during execution. Thus, the execution of a program consists of two separate parts. The first part initializes the environment, which includes creating all classes specified within the program, as well as the cache tables, et.al. The second part involves the actual execution of the modules. During the execution of this second part, new classes cannot be made.

5.3.3. makeMSPEnv

This routine initializes the environment. This consists of assigning the size variables *classNum*, *moduleNum*, *contextSize* and *numLiterals* the value 0, initializing diagnostics and memory management variables, parsing any flags from the command line, and creating the required classes and instances. All of these things are done in the same way no matter what program is run. Finally, the program specific environment information is obtained by calling the function *init_mspenv*, which is generated from ST80 code every time a new program is made. This function in turn initializes the selector index to color maps and the class index to class name map, creates and fills in the instance and class cache tables, executes the kernel module and creates any literal MSObjects specified within the program.

5.4. Code Generation

The PE class MSPEnv controls the generation of C-code. Upon receiving a request to generate C-code for a specific module (which defines a program), a sequence of actions occurs. First, the specified module and all modules recursively imported by it are collected. This collection will be referred to as the *minimal module set*. Next, a new cache table is created and all of the collected modules are parsed to initialize the new cache table. The result is a cache table which defines only those classes and selectors needed for the specified program (i.e. a minimal cache-table). The PE then generates a program file in C which initializes the C cache-table structure with the required information. Finally, the PE asks each module node class within the minimal module set to generate C-code for itself. Each module in turn generates C-macros explicitly, and implicitly by asking component node classes to generate C code for themselves. The result of this C generation is a source file for general program information (the main function and the cache table initialization), an associated header file, and a source file and header file for every module in the minimal module set. Below, the process described above is discussed in more detail.

5.4.1. Program Code Generation

A C source file is generated to hold the program functions. This includes the main function for the program, and the *init_mspenv* function mentioned previously. The name of this file is the concatentation of the main module name and the string "_program.c". Thus, for the example program, the file is named "Numbers_program.c".

The main program consists of macro calls to initialize global variables, initialize the MSPEnv, and executes the modules in the minimal module set *in the correct order*, such that a module is executed before it is imported by another module. The determination of order is done by the PE during code generation.

Figure 16 shows an algorithmic description of what the code generated for the *init_mspenv* function does.

```
INIT_DEF()
        initialize instance selector index to color static array
        initialize class selector index to color static array
        initialize class index to class name static array

        store selector index to color maps in MS environment structure
        store class index to class name map in MS environment structure

        Make instance cache table
        Make class cache table

        Make all literals specified in program
INIT_END();
```

Figure 16 : Algorithmic description of *init_mspenv* function

Note that the last thing the function does is make all literals specified in the program. The MS specification states that literals are immutable. This implies that it is possible (although not a requirement) that literals be unique. The current version takes this approach, and thus each literal is only created once and stored as part of the environment. Note that literals, being syntactic constructs, are identifiable at compile-time, so that all such literals can be created before execution of modules begins. In other words, the literals found in a program are identifiable from the MS program source code. Unique literals are discussed in more detail in Chapter 4.

5.4.2. Module Code Generation

All code needed to execute a module is placed in a C file, whose name is formed by appending a ".c" to the string representing the name of the module within the MS program. For example the C file storing the code for module *Numbers* is named *Numbers.c*.

Each module contains all the code needed to execute a module and all executable items declared within a module. C functions are used to execute class declarations, methods, literal blocks and method expressions.

A module is executed by a function whose name is formed by concentating the name of the module with "Module". Thus, for module *Numbers*, the name of the C function that executes it is *NumbersModule*.

Modules exist only within the environment (a MSPEnv C-structure), and the execution of a module results in a context that is inserted into the module array of the environment. The values of the context are MSObject * arrays, consisting of instances or classes, obtained via class definitions, modules expressions, or import statements. Note that the current implementation does not implement class extensions. Explicit module imports are treated as short-hand for implicit importation of every binding within the specified module.

```
MODULE_DEF( NumbersModule )
        ASSIGN_MODULE_SPACE( 3, "NumbersModule" );

        ASSIGN_MODULE_IMPORT( 0, 1, 0 );
        ASSIGN_MODULE_CLASS( 1, ComplexClass );
        ASSIGN_MODULE_EXPRESSION( 2, NumbersModule_E1 );
MODULE_END();
```

Figure 17 : Example Code Generated for Module Function

5.4.3. Class Definition Code Generation

C code generated for creating classes is placed in the module file to which the class belongs. The name of class functions consists of concentating the name of the class within the MS program with the string "Class". Thus, for a class called *Complex* within a module, the associated C function name is *ComplexClass*. Unless the ST80 dispatch mechanism is to be used, the only code generated within the C-function for classes is a macro calling the *makeClass* library function with appropriate arguments specifying the number of different types of variables. However, if a ST80 lookup mechanism is to be provided, code to create instance and class method dictionaries must also be specified. Figure 18 gives an example of generated C-code.

```
CLASSDEF_HEADER( ComplexClass )
        CLASSDEF_VARS( "ComplexClass", 5, 1 );
        CLASSDEF_MAKE( 2, 0, 0, 0, 0, 0, 23 );
CLASSDEF_END();
```

Figure 18 : Example Code Generated for Class Function

5.4.4. Method Code Generation

Method definitions are placed in the module C file in which they are defined. The code for them occurs after the code for their associated class. Method function names have a special format. Since C functions are not allowed to have ':' character or operator characters within them, a mechanism for generating an equivalent name for an MS program selector was used. Furthermore, since different classes can define the same selector, the name of the selector is not sufficient to uniquely identify the associated C function. Thus, all C function names start with the name of the class to which the current method belongs, followed by a "_", followed by the converted form of the selector. As examples, assuming the operator character '+' has been mapped to 'K' and the defining class is Complex, the C function names for instance selectors *real*, +, and class selector *real:imag:* are *Complex_real*, *Complex_binary_K*, and *Complex_class_realK_imagK* respectively.

The code generated for methods depends on the type of method. For undefined and abstract methods, the name of the C-function is renamed (using the C #define preprocessor command) to the primitive functions *undefinedMethod* and *abstractMethod* respectively. For primitive methods, the function is renamed to the name of the associated C primitive. The ST80 PE maintains implementations for all primitives, as well as the C function name of the primitive when generated in C. It is the responsibility of the programmer who makes MS primitives in the ST-80 environment to insure that the associated C function name is correctly specified. For aliased methods, the code generated is simply a preprocess define mapping the C function name to the aliased name specified.

The code for state methods depends on the kind. An accessing state method returns the value of the specified variable (or the size, in the case of an indexed size access state method). An assignment state method assigns a new value to the specified variable (or changes the size of the variable in the case of an indexed size assignment state method), and return the new value.

The code for block method definitions and for literals blocks within block methods is the same, both represented by individual C functions. The code generated first defines the function and three standard arguments (*rec*, *args* and *size*), and declares special local variables. The *result* (type MSObject *) variable stores the return value of the method. The *subBlockResult* (type MSObject *) variable stores the result of the last message send executed within the method, and will be described later. The *cntxt* variable (type MSContext *) represents the context of this method. After declaring these three variables, the context is initialized and pushed onto the dynamic context stack. How the context is initialized depends on whether the function is implementing a block method or a literal block. Note that even though literal blocks do not have names, and thus cannot be called directly, they can still exist on the dynamic stack more than once at the same time because the block method in which they reside can be called more than once. The static link of a block method's context is set to the context of the module in which the block method is defined. The static link of literal block contexts will be discussed later along with the concept of closures.

After the set-up macro code above, C code is generated for each statement within the method. The code generated for each type of statement is discussed below. An example of generated C code for a entire method is given in Figure 19.

```
BLOCKDEF_HEADER( Complex_binary_K  /* + */ )
        BLOCKDEF_VARS( "Complex_binary_K  /* + */", 1, 1, 3, BLOCK_IS_METHOD );

        MESS_SEND( 2 );
                MESS_REC() = CNTXT_OFFSET( CC, 1 );  /* CNTXT ( -1, 1) */
                SEND_MESSAGE( 2, -1, 2 );  /* real:imag: */
        MESS_END( 1 );
        CNTXT_OFFSET( CC, 2 ) = MESS_RESULT();

        BLOCK_RESULT() = CNTXT_OFFSET( CC, 2 );  /* CNTXT ( 0, 2) */
        BLOCKDEF_EXPLICIT_RETURN( BLOCK_IS_METHOD );
BLOCKDEF_END();
```

Figure 19 : Example Code Generated for Block Method

5.4.5. Code Generation for Variables, Self and Literals

Code generated for variables is a macro accessing the proper level and offset within the current context. The pseudo variable *self* is always stored as the object at offset 0 of all contexts, so the same context macro, with a level and offset of 0, can be used to obtain the proper (receiver) object. Code for a literal consists of a macro accessing a specific index in the literal hash table.

5.4.6. Code Generation for Literal blocks

Literal blocks within a parent block are represented by another C function This is always necessary when the literal blocks are arguments to a message send, since the block is not necessarily executed within the method in which it syntactically appears. Note that it is possible to generate C code for non-argument literal blocks in-line within the function of the parent block, but the current implementation does not do this.

For literal blocks whose parent is a block method, the name of the function implementing the literal block consists of the name of the block method function, followed by "_block_" followed by an integer, starting at 1, and incrementing by one for every literal block at the same level. Thus, if there are two literal blocks within a block method whose function name is *test*, then the names of the functions implementing the two literal blocks are *test_block_1"* and *test_block_2"*. The order of the numbers is specified by the syntactic order of the blocks within the program.

For literal blocks whose parent is a literal block, the name of the function is the name of the parent block function, followed by "_", followed by an integer starting at 1, with the same semantics as above.

The code generated within a parent block when a literal block is encountered consists of a macro to create a new closure object. A *closure* is an MSObject instance of the MS class Closure, consisting of a function address and a context. The current implementation does not handle literal blocks that are not arguments to a message send.

5.4.7. Code Generation for Message Sends

An MSMessageSend parse node generates C-code that uses C-subblocks. Subblocks are required because the arguments to message sends can themselves be arbitrarily complex message sends. Certain variables must be maintained during the execution of the current message send, and it is desirable if every message send uses a consistent format (to make code generation possible). The C language has subblocks, in which variables can be defined and assigned values, such that if the variable existed in an outer scope, its value within the subblock does not affect the value it had outside the subblock. Although access to the variable in the outer block is not possible from within the subblock (they both have the same name, and any reference to that name references the variable associated with the current block), this does not pose a problem for our purposes, since nested message sends do not need to know any information of messages sends above them. This subblock facility thus provides the ability to nest blocks arbitrarily deeply, providing for arbitrarily deep message send nesting.

With this in mind, C code generation for messages sends proceeds as follows. A macro is generated to create a subblock and define two local variables, *rec* (an MSObject *) and *args* (array of MSObject *). The *args* variable is initialized to a dynamically created array whose size equals the number of arguments to the message send. Macros to assign appropriate argument objects to the args array and to the receiver are then generated. Note that the code so generated may be arbitrarily complex, involving many nested message sends, assignments, variable accesses, etc. After the args and rec variables have been assigned values, C code is generated to actually perform the message send. The *execute* function is called which accesses the cache table, finds the desired method, executes it and returns a result object, which is assigned to the variable *subBlockResult*. Remember that *subBlockResult* was a variable defined at the beginning of the C function for the block in question. Finally, a macro to check for explicit returns and to close the subblock is generated.

In order to understand the code generated for an arbitrarily complex message send, in which arguments are also message sends, it is best to first look at the MS source code and resulting C code generated for a simply message send whose arguments do not perform message sends.

```
/*
 For an MS message send like :

    self at: t put: 3

 Where it is assumed that the message occured within a literal block
   within a block method, that the literal block has one temporary variable
   called 't', and that there are at least 2 literals in the program,
   with the MSObject representing the integer literal '3' being stored
   in index 1 of the MS.literal array.
 */

MESS_SEND( 2 );
        MESS_ARG( 0 ) = CNTXT_OFFSET( CC, 0 );  /* CNTXT ( 0, 0) */
        MESS_ARG( 1 ) = LITERAL_AT_INDEX( 1 );   /* 3 */
        MESS_REC() = CNTXT_OFFSET( CN( CC ), 0 );  /* CNTXT ( 1, 0) */
        SEND_MESSAGE( 2, 23, -1 );  /* at:put: */
MESS_END( 0 );
```

Figure 20 : Example Code Generated for Simple Message Send

In the example, the *MESS_SEND(numArgs)* macro specifies that a message send with two arguments is coming. The rec and args variable are defined in a subblock, and the args array is dynamically allocated. Next, the args array is initialized. Index 0 is initialized to the value of the variable whose level is 0, and whose offset is 0. This refers to the variable 't'. Index 1 is assigned the MSObject representing the literal 3, since it is assigned the value of index 1 of the environment literal array. Next, the *rec* variable is assigned. The C macro *CNTXT_OFFSET(CN(CC), 0)* specifies that the variable with level 1 and offset 0 is desired (the level is determined by counting the number of *CN()* macros in the expression). This code says to obtain the static link of the current context and obtain offset 0 from the data array of this context. Since level 1 for this example refers to the block method, offset 0 is the pseudo variable 'self'. The *SEND_MESSAGE* executes the *execute* function, which requires a receiver (rec), an array of arguments (args), the number of args (2), the instance selector index of desired message to send (23), and the class selector index of desired message to send (-1). The result of calling the function is assigned to the variable *subBlockResult*. Finally, the *MESS_END()* macro frees the local args array, checks to see if the result of the message send is a return object. Since, according to the example code, it is impossible for it to be a return object, nothing more is done. The macro generates a close brace to finish the subblock.

Since the result of a message send is stored in the variable *subBlockResult*, it is possible to assign an argument index to the result of a message send by first generating the code for the sub message send, then assigning the appropriate argument index to the *subBlockResult* variables. The macro used to access the *subBlockResult* variable is *MESS_RESULT()*.

```
/*
 For an MS message send like :

   0 to: t - 1 do: [ "anything" ]

  Where the message send occurs in a literal block inside a block method,
   t is the first variable declared in the block method, and the literal
   array contains an MSobject representing '0' at index 0 and '1' at index 1.

*/
MESS_SEND( 2 );
        MESS_SEND( 1 );
                MESS_ARG( 0 ) = LITERAL_AT_INDEX( 1 );    /* 1 */
                MESS_REC() = CNTXT_OFFSET( CN( CC ), 1 );  /* CNTXT ( 1, 1) */
                SEND_MESSAGE( 1, 16, -1 );  /* - */
        MESS_END( 0 );
        MESS_ARG( 0 ) = MESS_RESULT();
        ASSIGN_NEW_CLOSURE( MESS_ARG( 1 ), A_initializeK_iterationsK_block_1, 1, CC );
        MESS_REC() = LITERAL_AT_INDEX( 0 );    /* 0 */
        SEND_MESSAGE( 2, 19, -1 );  /* to:do: */
MESS_END( 0 );
```

Figure 21 : Example Code Generated for Complex Message Send

In the example above, code to define a subblock and allocate a 2 index args array and define a rec variable is specified. Since the first argument to the message send is itself a message send, a subblock is made within the first subblock which allocates a 1 index args array and defines a rec variable. Index 0 of this args array is set to the literal at index 1 of the global literal array. The rec variable is set to a variable reference which has level 1 and offset 1. The message is executed, the args array is freed and the submessage send is finished. Since the subsubblock has closed, the values of the args and rec variable are back to what they were before the subsubblock was defined (i.e. the args array is of size 2). Index 0 of this args array is set to the result of the sub message send. Index 1 is set to a closure representing a literal block. The rec variable is set to the literal object '0', and this message is sent of for execution. Finally, the args array is freed and the message send is finished.

5.4.8. Code Generation for Assignments and Returns

Since the left hand side of an assignment is always a variable reference, it will always be represented in C as variable access code as described above. The right hand side will be represented by one of the macros already described. An assignment cannot have an MSCReturn as its expression since the compiler would have generated an error.

An MSReturn parse node must generate code to return immediately whatever its associated expression is. It generates an C code assignment of its associated expression to the result variable, then generates a macro which sets the explicit return flag and immediately returns from the C function.

```
/* For the MS code

     ^self

  The following is generated
*/
BLOCK_RESULT() = BLOCK_SELF();
BLOCKDEF_EXPLICIT_RETURN( BLOCK_METHOD );
```

Figure 22 : Example Code Generated for Explicit Return

5.5. Optimizations

To increase the efficiency of the compiler version of a MS program, various optimizations are possible. Discussed below are two primary areas in which optimization has promise.

5.5.1. Method Dispatch

At the time that C-Code generation occurs, the ST-80 MS Environment knows which selectors are uniquely defined. These are the selectors whose divisions are specific or whose receiver is a literal or self. In these cases, message sends can be optimized - the execute function does not need to be called to obtain the function. Instead, where the execute would have occurred, a direct call to the appropriate function can be made. That is, a macro can be generated that pushes the appropriate arguments onto the stack and performs this direct call.

This optimization can be extended in the case of state accessing methods if self is the receiver. In this case, no function call is needed at all — the code to access the desired variable can be placed in-line where the message would have been called.

If static typing is added to the language, the amount of lookup required will also be reduced, since more selectors will be uniquely defined. Even if the selector isn't unique, polymorphic in-line caches will be more efficient than look-up, provided that the number of different methods is small enough.

5.5.2. Literal Blocks and Module Expressions.

Inefficiency exists whenever a function call is made. Currently, literal blocks are always represented as independent C functions. However, if these literal blocks are not arguments to a message send, and are instead stand-alone subblocks within a method, the code for them can be imbedded directly within the code for the parent block. However, it is suspected that this optimization will provide very little speed-up unless the method in question is called many times. Furthermore some complexity exists in implementing the generation this code in-line.

A related optimization involves module expressions, which are also currently implemented by dedicated functions. Code generated directly within the function for the module would provide some improvement. However, even more so than for literal blocks, the amount of speed improvement is questionable.

Chapter 6 : Future Directions

6.1. Conclusions

This thesis presented a first implementation of Modular Smalltalk that validates two of the language design goals: consistent execution semantics and efficient code execution We intend to use this implementation to validate the other three design goals: increased programmer productivity through code reuse and code re-definition, design and implementation efficiency for multiple programmer applications and simplicity for new users.

The implementation is an object-oriented one and illustrates the use of an object-oriented approach to: parsing, program representation and code generation. The code generation is modular and based on macros so it can be easily modified to support a variety of target languages including C and assembly language. The efficient implementation includes a cache table approach to method dispatch that uses extensions to the André-Royer incremental coloring algorithm. The new coloring algorithm includes support for optional static typing and we are currently studying the effects of typing and multiple cache tables on dispatch efficiency. Currently, we have only implemented a simple reference-counting storage manager. We intend to incorporate a more efficient scavenging approach [UJ88] [WM89]. Our implementation will be available by anonymous ftp when the documentation is complete.

6.2. Future Directions

There are many different areas in which future research is planned.

6.2.1. Static Typing

One of the more important planned modifications to the language consists of introducing optional static typing. The ICP dispatch algorithm maintains information which, when combined with static typing, can reduce the amount of lookup needed to obtain methods. Because method lookup is such a large part of the efficiency issue, static typing should give good results in providing more efficiency. Furthermore, more compile time error checking can be performed.

6.2.2. Software Verifiability

As another extension to the language, providing mechanisms similar to Eiffel such as pre and post conditions and assertions will provide for software verifiability.

6.2.3. Cache Tables

The current implementation uses one cache table for all instance behavior in the environment, and another for all class behavior in the environment. The advantages of this two table approach over a single table approach were discussed in Chapter 4, primarily involving space savings.

If static typing exists in the language, the same type of space savings can also be had by dividing each of the two cache tables into multiple tables, one table for each tree in the hierarchy forest.

More research on the relevant advantages and disadvantages of these approaches is needed, both from an implementation complexity stand point, and a space efficiency one. Note that all approaches will have very close to the same runtime execution efficiency.

6.2.4. Garbage Collection

The current implementation of the PE does not have any garbage collection, and for the compiled code, a simple reference count garbage collector is used. In both cases, efficient versions based on generational scavenging will be implemented.

6.2.5. MS Library and Automatic ST-MS Converter

During the creation of the language, an automatic Smalltalk to MS converter was written. This, coupled with small changes to the MS parser, provides for an efficient method of obtaining MS code from Smalltalk code.

The current MS library consists of basic behavior for a Object class and its superclasses, trivial behavior and appropriate state for the required classes, and a partial implementation of the Collection hierarchy, refactored to use multiple inheritance to its best advantage.

Future work will extend this library. The minimum acceptable library must have a working collection hierarchy and I/O in the form of a Stream hiearchy.

6.2.6. STACC : Automated Parsing Framework and Generator

During the implementation of the parsing code, it was recognized that an automated mechanism for doing the work would be relatively easy to create. The result is a generator of node classes capable of forming an internal representation of any BNF specifiable language [SH93]. This generator only requires augmented BNF grammar rules to describe the tokens and grammar rules needed in order to generate all node classes needed. Additional behavior can then be added to provide for execution, other-language generation, decompilation, etc.

The limit on the amount of behavior that can be automatically generated has not yet been exhausted. For example, with minimal additional information, decompilation should be automatable.

6.2.7. Self-Implementation

Having multiple inheritance during the implementation of the PE would have made the internal representation much cleaner. Creating the current PE, but written in MS, would provide such a mechanism, and should be quite straightforward. However, there are distinct disadvantages — currently one can use all of the graphical debugging facilities of ST80. Self-implementation would imply that many complex Smalltalk functions would have to be written to provide the same graphical environment.

Bibliography

[AR92] P. André and J. Royer. "Optimizing Method Search with Lookup Caches and Incremental Coloring." In *OOPSLA '92 Conference Proceedings*, pp.110-126, October 1992.

[Coo92] W. Cook. "Interfaces and Specifications for the Smalltalk-80 Collection Classes." In *OOPSLA '92 Conference Proceedings*, pp.1-15, October 1992.

[CU91] C. Chambers and D. Unger. "Making Pure Object-Oriented Languages Practical." In *OOPSLA '91 Conference Proceedings*, pp.1-15, October 1991.

[DMSV89] R. Dixon et. al. "A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance." In *OOPSLA '89 Conference Proceedings*, pp.211-214, October 1989.

[GR89] A. Goldberg and D. Robson. *ST-80, The Language*, Addison-Wesley, Reading Mass, 1989.

[Tek89] Tektronix. *Modular Smalltalk Language Proposal*, Technical Report CRL-89-03, 1989.

[UJ88] D. Unger and F. Jackson. "Tenuring Policies for Generation-Based Storage Reclamation." In *OOPSLA '88 Conference Proceedings*, pp.1-17, September 1988.

[WBW88] A. Wirfs-Brock and B. Wilkerson. "An Overview of MS". In *OOPSLA '88 Conference Proceedings*, pp.123-134, September 1988.

[WM89] P. Wilson and T. Moher. "Design of the Opportunistic Garbage Collector". In *OOPSLA '89 Conference Proceedings*, pp.23-35, October 1989.

[SH93] D. Szafron and W. Holst. "Automated Parsing Framework using BNF Grammars". Unpublished

[HS93] W. Holst and D. Szafron. "Efficient Object-oriented Method Dispatch". Unpublished

Appendix A : BNF Syntax Reference

In the following BNF grammar rules, expressions enclosed in square brackets([...]) may occur zero or one times. Expressions enclosed in curly brackets({...}) may occur zero or more times. All characters or strings (i.e. literal tokens) of characters appearing in the language are enclosed in single quotes ('...'). Nonprinting ASCII characters are given by using the associated C language backslash convention (i.e. for tab).

The following is a description of the actual token classes used in the PE.

```
>Integer<            ::= ['-'] <digits> | <radix specifications> ['-'] <based digit> { <based digit>}
>LiteralSelector<    ::=  '#' <method selector>
>Float<              ::= ['-'] <digits> <sub floating point>
>Character<          ::=  '$' <printing character>
>Array<              ::=  '#(' {<literal>} ')'
>String<             ::=  '"' { <literal character> | <separator character> |  '"quote""quote"' | '""'  } '"'
>KeywordSelector<    ::= <identifier> ':' { <identifier> ':' }
>BinarySelector<     ::= <operator character> {<operator character>}
>Declarer<           ::=  '->'
>VisPublic<          ::=  '(public)'
>VisPrivate<         ::=  '(private)'
>TypeVariable<       ::=  'variable'
>TypeBinary<         ::=  'binary'
>Word<               ::= <letter> { <letter> | <digit> }
```

Figure 23 : BNF Grammar Rules for Parse Tree Token Classes

The following is a description of the actual node classes used in the PE. An augmented BNF grammar format provides all information necessary to automate the process of creating all token classes, node classes and corresponding parsing behavior.

Within a rule, tokens are denoted by

>TokenName(state)<

and node classes are denoted by

<NodeName(state)<

The *state* referred to above is used by the automated code to determine which state methods to use when adding a token/node instance to the current parsing node. This state is not needed to understand the syntax of the MS language and can be ignored.

```
<Module>            ::= ‘{’ ‘module’ [>String(name)<] {<Binding(addBinding)>} ‘}’
<Binding>           ::= >Word(name)< [>VisAttr(privacy)<] ‘->’ (<Import(value)> | <ClassDefn(value)> |
                        <ModExpr(value)>)
<Import>            ::= ‘{’ [ ‘import’ >Word(bindingName)< ] ‘from’ >String(moduleName)< ‘}’
<ModExpr>           ::= ‘{’ ‘expression’ <MessageSend(expression)> ‘}’
<ClassDefn>         ::= ‘{’ ‘class’ [ ‘{’ ‘refines’ { >Word(addSuperclass)< } ‘}’ ] [ ‘instance’ <Behavior(instBehavior)> ]
                        [ ‘class’ <Behavior(classBehavior)> ] ‘}’
<Behavior>          ::= ‘{’ ‘behavior’ { <StateMethod(addStateMethod)> | <Method(addMethod)> } ‘}’
<StateMethod>       ::= ‘{’ ( ( >UnarySelector< [>VisAttr<] >SingleKeywordSelector< [>VisAttr<] ) |
                        >SingleKeywordSelector< [>VisAttr<] >UnarySelector< [>VisAttr<] ) ‘}’ ‘->’ ‘variable’ ) | (
                        ( >UnarySelector< [>VisAttr<] >SingleKeywordSelector< [>VisAttr<] |
                        >SingleKeywordSelector< [>VisAttr<] >UnarySelector< [>VisAttr<] ) ‘|’ (
                        >SingleKeywordSelector< [>VisAttr<] >DoubleKeywordSelector< [>VisAttr<] | >Doub-
                        leKeywordSelector< [>VisAttr<] >SingleKeywordSelector< [>VisAttr<] ) ‘}’ ‘->’ ( ‘variable’
                        | ‘binary’ ) ) )
<Method>            ::= >Selector< [ >VisAttr< ] ‘->’ ( [ ‘method’ ] ( <Block(definition)> |
                        <PrimitiveMethod(definition)> | <UndefMethod(definition)> |
                        <AbstractMethod(definition)> | <AliasMethod(definition)> )
<AliasMethod>       ::= alias’ >Word(className)< >Selector(methodName)<
<PrimitiveMethod>   ::= ‘primitive’
<AbstractMethod>    ::= ‘abstract’
<UndefMethod>       ::= ‘undefined’
<Block>             ::= ‘[’ [ { ‘:’ >Word(addParameter)< } ‘|’ ] [ ‘|’ { >Word(addTemporary)< } ‘|’ ] { ( <Assignment>
                        | <MessageSend> ) ‘.’ } [ ( ( <Assignment> | <MessageSend> ) | ‘^’ ( <Assignment> |
                        <MessageSend> ) ) [ ‘.’ ] ]
<Assignment>        ::= [ >Word(addVariable)< ‘:=’ ] <MessageSend(statement)>
<MessageSend>       ::= ( >Word(receiver)< | <Literal(receiver)> | <Block(receiver)> | <Expression(receiver)> ) [
                        <Message(addMessage)> { ‘;’ <Message(addMessage)> } ]
<Message>           ::= very complex
<Expression>        ::= ‘(’ ( <Assignment(statement)> | <MessageSend(statement)> ) ‘)’
<Literal>           ::= >Literal(state)<
```

Figure 24 : BNF Grammar Rules for Parse Tree Node Classes

The following productions describe the syntax of Modular Smalltalk as it was given in the original specification.

Productions have been grouped into sections for the reader’s convenience.

Character Class Productions

```
<digit>                ::= ´[0-9]´
<letter>               ::= ´[a-zA-Z]´
<character>            ::= <printing character> | <separator character>
<printing character>   ::= <nonquote character> | ‘´’ | ‘"’
<nonquote character>   ::= <letter> | <digit> | <operator character> | <special character>
<operator character>   ::= ´[+-*/<=>~&|,@?%!‘]´
<special character>    ::= ´[()[]{}#$.;:^_]´
<separator character>  ::= ´[ \t\r\n]´
```

Literal Productions

<literal> ::= <numeric literal> | <literal character> | <string> | <literal selector> | <array>
<numeric literal> ::= <integer> | <floating point>
<integer> ::= ['-'] <digits> | <radix specifications> ['-'] <based digit> { <based digit>}
<digits> ::= <digit> {<digit>}
<radix specifications> ::= <digits> 'r'
<based digit> ::= <digit> | <letter>
<floating point> ::= ['-'] <digits> <sub floating point>
<sub floating point> ::= <fraction part> | <fraction part> <exponent part> | <exponent part>
<fraction part> ::= '.' <digits>
<exponent part> ::= <sub exponent part> ['-'] <digits>
<sub exponent part> ::= 'e' | 'E'
<literal character> ::= '$' <printing character>
<string> ::= ''' { <sub string> } '''
<sub string> ::= <literal character> | <separator character> | '"quote""quote"' | "'
<literal selector> ::= '#' <method selector>
<method selector> ::= <unary selector> | <binary selector> | <keyword selector> {<keyword selector>}
<unary selector> ::= <identifier>
<identifier> ::= <letter> { <letter> | <digit> }
<binary selector> ::= <operator character> {<operator character>}
<keyword selector> ::= <identifier> ':'
<two arg keyword selector> ::= <keyword selector> <keyword selector>
<array> ::= '#(' {<literal>} ')'

Separator Productions

<separator> ::= { <separator character> | <comment> }
<comment> ::= '"' { <nonquote character> | '' | <separator character> } '"'[1]

Expression Productions

<expression> ::= { <variable> ':=' } <message send>
<message send> ::= <primary> [<message pattern> { ';' <message pattern>}]
<primary> ::= <name> | <literal> | '(' <expression> ')' | <block>
<name> ::= <module constant> | <parameter> | <variable>
<module constant> ::= <constant name> | <class name>
<constant name> ::= <identifier>
<class name> ::= <identifier>
<parameter> ::= <identifier>
<variable> ::= <identifier>
<message pattern> ::= <unary message> { <unary message> } {<binary message>} [<keyword message>] | <binary message> { <binary message> } [<keyword message>] | <keyword message>
<unary message> ::= <unary selector>
<binary message> ::= <binary selector> <primary> { <unary message> }
<keyword message> ::= <keyword part> { <keyword part> }
<keyword part> ::= <keyword selector> <primary> { <unary message> } { <binary message> }

---

[1] Erroneous in [TEX89], since it requires that every character be prepended with a '$'.

Method Declaration Productions

<method declaration> ::= <state declaration> | <method>
<state declaration> ::= <named state decl.> | <indexable state decl.>
<named state decl.> ::= '{' <state pair> '}' <is declared as> 'variable'
<state pair> ::= <access selector> <change selector> | <change selector> <access selector>
<is declared as> ::= '->'
<access selector> ::= <unary selector> [<visibility attribute>]
<change selector> ::= <keyword selector> [<visibility attribute>]
<indexable state decl.> ::= '{' <state pair> '|' <indexing pair> '}' <is declared as> <sub isd>
<sub isd> ::= 'variable' | 'binary'
<indexing pair> ::= <indexed access sel.> <indexed change sel.> | <indexed change sel.> <indexed access sel.>
<indexed access sel.> ::= <keyword selector> [<visibility attribute>]
<indexed change sel.> ::= <two argument keyword selector> [<visibility attribute>]
<method> ::= <method selector> [<visibility attribute>] <is declared as> <method definition>
<visibility attribute> ::= '(private)' | '(public)'
<method definition> ::= <block declaration> | <method alias> | <primitive declaration> | <abstract declaration> | <undefined declaration>
<block declaration> ::= ['method'] <block>
<block> ::= '[' [{ ':' <parameter> } '|'] <code body> ']'
<code body> ::= ['|' { <variable> } '|'] <expression list>
<expression list> ::= { <expression> '.' } [<sub el>]
<sub el> ::= <expression> | '^' <expression> ['.']
<method alias> ::= 'alias' <class name> <method selector>
<primitive declaration> ::= 'primitive'
<abstract declaration> ::= 'abstract'
<undefined declaration> ::= 'undefined'

Class Declaration Productions

<class declaration> ::= '{' 'class' [<class interface>] [<instance behaviour declaration>] [<class behaviour declaration>] '}'
<class interface> ::= '{' 'refines' { <class name> } '}'
<instance beh. decl.> ::= 'instance' <behaviour declaration>
<class beh. decl.> ::= 'class' <behaviour declaration>
<behaviour decl.> ::= '{' 'behaviour' { <method declaration> } '}'

Module Declaration Productions

<module declaration> ::= '{' 'module' [<module name>] { <declaration> } '}'
<module name> ::= <string>
<declaration> ::= <binding> | <class extension> | <module import>
<binding> ::= <identifier> [<visibility attribute>] <is declared as> <binding declaration>
<binding declaration> ::= <import declaration> | <class declaration> | <expression declaration>
<import declaration> ::= '{' ['import' <identifier>] 'from' <module name> '}'
<expression decl.> ::= '{' 'expression' <message send> '}'
<class extension> ::= '{' 'extend' <class name> [<instance behaviour declaration>] [<class behaviour declaration>] '}'
<module import> ::= '{' 'use' <module name> '}'

Appendix B : Method Dispatch Algorithms

The following definitions extend the definitions given in the ICP algorithm discussion:

Symbol	**Definition**
executeMethod(D)	return the result of executing the method for division D
divisionAt[T,C,L]	entry in table T at class index of C and color index L
isClass(O)	return true if object O is a class.
divisionEmpty(D)	if division D isKindOf: MSEmptyDivision, true else false
classMethods(C)	hash table of all class methods (as divisions)
instanceMethods(C)	hash table of all instance methods (as divisions)
selectorPrivateInClass(C,S)	[if selector S is private in class C, true else false]
maxColorStoredForClass(C,T)	maximum color index stored for class C in table T.
colorStoredForClass(L,C,T)	[L <= maxColorStoredForClass(C, T)]

```
execute(receiver : Object; args : Object; size : Integer; instSelector,
        classSelector : Selector) : Object

begin
        if isClass( receiver ) then
                set S = classSelector
                set T = class table
        else
                set S = instSelector
                set T = inst table
        endif

        set D = divisionAt[ T, class(receiver), color(S) ]

        if ((colorStoredForClass(color(S), class(receiver), T) and
                (divisionSelector(D) == S)) then
                set result = executeMethod( D )
        else
                generate messageNotUnderstood error.
        endif
  return result
end
```

Figure 25 : Method Dispatch Algorithm For MS using Two Cache Tables

```
execute( receiver : Object; args : Object; size : Integer )
          instSelector, classSelector : Selector ) : Object

begin
  set D = lookup_dispatch(receiver, class(receiver), instSelector, classSelector)

  if (not divisionEmpty(D))
    set result = executeMethod(D)
  else
    generate messageNotUnderstood error.
  endif
          return result
end


lookup_dispatch(receiver: Object; class: Object; instSelector, classSelector :
          Selector) : Division

begin
   if isClass(receiver) then
      set selector = classSelector
      set methodDictionary = classMethods(class)
   else
      set selector = instSelector
      set methodDictionary = instanceMethods(class)
   endif

   find bucket K for selector S in methodDictionary hash.
   if K includes S,
      set D = division stored for S in K.
      if (selectorPrivateInClass(class(receiver), S)) then
         if (class(receiver) != class) then
            set D = emptyDivision
         endif
      endif
   else
      set D = emtpyDivision
   endif

   if (divisionEmpty(D)) then
      loop over superclasses(C)
         set D = lookup_dispatch(receiver, newClass)
      until ((not divisionEmpty(D)))
   endif

   return D
end
```

Figure 26 : Class Look-up Dispatch Algorithm for Multiple Inheritance

Appendix C : Sample of Generated C Code

This appendix lists the C macro code generated for the Numbers Module given in Chapter 3. After this code is the actual C code (macro expanded). Anytime /* ... */ is seen in the text, information has been removed to save space.

The code below represents the program file generated to create the cache-tables, execute the modules, etc.

```
# include "cgen_lib.h"
# include "Numbers_all_modules.h"

DECLARE_GLOBALS();

PROGRAM_MAIN()
  PROGRAM_INIT( "Numbers" );

  EXECUTE_MODULE( NumbersModule, 3 );

PROGRAM_END();

INIT_DEF()
   INSTCOLORS()
      45 , 44 , 41 , 42 , -1 , 36 , 37 , -1 , 40 , 8  ,
      /* ... */
   END_COLORS();

   CLASSCOLORS()
      0  , 1  , 1
   END_COLORS();

   CLASSNAMES()
      "MemoryObject",        /* 0 */
      "TestableObject",      /* 1 */
      "ComparableObject",        /* 2 */
      /* ... */
      "Complex",         /* 23 */
   END_NAMES();

   INIT_ENV_NAMES();

   INIT_ENV_COLORS();

   TABLE_NEW( INST_TABLE, 24 );
   /* ... */

   TABLE_NEWAT( INST_TABLE, 10, 36 );
   /* ... */
   TABLE_ATPUT( INST_TABLE, 10, 19, 41, PrintableObject_printOnK, "PrintableObject_printOnK" );
   TABLE_ATPUT( INST_TABLE, 10, 20, 27, ClassableObject_species, "ClassableObject_species" );
   TABLE_ATPUT( INST_TABLE, 10, 21, 37, ErrorHandlingObject_errorK, "ErrorHandlingObject_errorK" );
   /* ... */

   TABLE_NEWAT( INST_TABLE, 23, 48 );
   TABLE_ATPUT( INST_TABLE, 23, 35, 30, ClassableObject_respondsToK, "ClassableObject_respondsToK" );
   TABLE_ATPUT( INST_TABLE, 23, 36, 45, Complex_binary_K  /* +  */, "Complex_binary_K  /* +  */" );
TABLE_ATPUT( INST_TABLE, 23, 37, 79, Complex_real, "Complex_real" );
TABLE_ATPUT( INST_TABLE, 23, 38, -1, NO_FUNCTION, "NO_FUNCTION" );
TABLE_ATPUT( INST_TABLE, 23, 39, 81, Complex_imag, "Complex_imag" );
/* ... */

TABLE_NEW( CLASS_TABLE, 24 );

TABLE_NEWAT( CLASS_TABLE, 0, 0 );
/* ... */

EXECUTE_MODULE( ObjectModule, 0 );
EXECUTE_MODULE( KernelModule, 1 );

LITERAL_ASSIGN_SPACE( 70 );

/* ... */
LITERAL_SET_INDEX( 65,
    ASSIGN_NEW_SIMPLE_LITERAL( LITERAL_TMP, GO_CLASS_INTEGER, 0) );
LITERAL_SET_INDEX( 66,
    ASSIGN_NEW_SIMPLE_LITERAL( LITERAL_TMP, GO_CLASS_INTEGER, 1) );
/* ... */
```

The following shows sample code generated for a module. The C-code generation used the decompile facility to automatically generate the MS Source code listing associated with various blocks of code (each class definition, for example).

```
/* This file implements the creation of all classes in the
 Numbers module, as well as the Numbers module itself. */

# include "Numbers.h"
# include "cgen_lib.h"


/***
 ***   Complex
 ***/

/*
    class
    { refines Object }

    instance
    { behaviour
        { imag imag: } -> variable
        { real real: } -> variable

        asReal -> method
        [self imag == 0
            ifTrue: [^self real .]
            ifFalse: [^nil .].]

        * -> primitive

        + -> method
        [ :aComplex |
            | result |
            result := Complex
                real: (self real + aComplex real )
                imag: (self imag + aComplex imag ).
            ^result .
        ]

        isNil -> alias Object isNil

        dummy -> abstract

        < -> undefined
    }

    class
    { behaviour
        real:imag: -> method
        [ :real :imag | self real: real ;
             imag: imag .]
    }
 */

CLASSDEF_HEADER( ComplexClass )
    CLASSDEF_VARS( "ComplexClass", 10, 1 );

# ifdef ST80_DISPATCH
    CLASSDEF_INIT_HASH( 10, 1, 1 );
    CLASSDEF_ADD_SUPERCLASS( 10 );

    CLASSDEF_SET_INST( 0, 11, Complex_isNil, VIS_PUBLIC );
    CLASSDEF_SET_INST( 1, 45, Complex_binary_K   /* +  */, VIS_PUBLIC );
    CLASSDEF_SET_INST( 2, 47, Complex_binary_J   /* *  */, VIS_PUBLIC );
    CLASSDEF_SET_INST( 3, 50, Complex_binary_Q   /* <  */, VIS_PUBLIC );
    CLASSDEF_SET_INST( 4, 76, Complex_dummy, VIS_PUBLIC );
    CLASSDEF_SET_INST( 5, 79, Complex_real, VIS_PUBLIC );
    CLASSDEF_SET_INST( 6, 80, Complex_realK, VIS_PUBLIC );
    CLASSDEF_SET_INST( 7, 81, Complex_imag, VIS_PUBLIC );
    CLASSDEF_SET_INST( 8, 82, Complex_imagK, VIS_PUBLIC );
    CLASSDEF_SET_INST( 9, 83, Complex_asReal, VIS_PUBLIC );

    CLASSDEF_SET_CLASS( 0, 2, Complex_class_realK_imagK, VIS_PUBLIC );
# endif

    CLASSDEF_MAKE( 2, 0, 0, 0, 0, 0, 23 );
CLASSDEF_END();

BLOCKDEF_HEADER( Complex_binary_K   /* +  */ )
    BLOCKDEF_VARS( "Complex_binary_K   /* + */", 1, 1, 3, BLOCK_IS_METHOD );

    MESS_SEND( 2 );
        MESS_SEND( 1 );
            MESS_SEND( 0 );
                MESS_REC() = CNTXT_OFFSET( CC, 1 );  /*  CNTXT ( 0, 1)  */
                SEND_MESSAGE( 0, 79, -1 );  /* real */
            MESS_END( 1 );
            MESS_ARG( 0 ) = MESS_RESULT();
            MESS_SEND( 0 );
                MESS_REC() = CNTXT_OFFSET( CC, 0 );  /*  CNTXT ( 0, 0)  */
                SEND_MESSAGE( 0, 79, -1 );  /* real */
            MESS_END( 1 );
            MESS_REC() = MESS_RESULT();
            SEND_MESSAGE( 1, 45, -1 );   /* + */
        MESS_END( 1 );
        MESS_ARG( 0 ) = MESS_RESULT();
        MESS_SEND( 1 );
            MESS_SEND( 0 );
                MESS_REC() = CNTXT_OFFSET( CC, 1 );   /*  CNTXT ( 0, 1)  */
                SEND_MESSAGE( 0, 81, -1 );   /* imag */
            MESS_END( 1 );
            MESS_ARG( 0 ) = MESS_RESULT();
            MESS_SEND( 0 );
                MESS_REC() = CNTXT_OFFSET( CC, 0 );   /*  CNTXT ( 0, 0)  */
                SEND_MESSAGE( 0, 81, -1 );   /* imag */
            MESS_END( 1 );
            MESS_REC() = MESS_RESULT();
            SEND_MESSAGE( 1, 45, -1 );   /* + */
        MESS_END( 1 );
        MESS_ARG( 1 ) = MESS_RESULT();
        MESS_REC() = CNTXT_OFFSET( CC, 1 );   /*  CNTXT ( -1, 1)  */
        SEND_MESSAGE( 2, -1, 2 );   /* real:imag: */
    MESS_END( 1 );
    CNTXT_OFFSET( CC, 2 ) = MESS_RESULT();

    BLOCK_RESULT() = CNTXT_OFFSET( CC, 2 );   /*  CNTXT ( 0, 2)  */
    BLOCKDEF_EXPLICIT_RETURN( BLOCK_IS_METHOD );
BLOCKDEF_END();
/* Primitive 'Complex_binary_J   // *  // */

STATEMETHOD_HEADER( Complex_real );
    STATEMETHOD_ACCESSNAMED( "Complex_real", OBJ_NAMED_AT( BLOCK_SELF(), 0 ) );
STATEMETHOD_END();

STATEMETHOD_HEADER( Complex_realK );
    STATEMETHOD_CHANGENAMED( "Complex_realK", OBJ_NAMED_AT( BLOCK_SELF(), 0 ), MESS_ARG(0) );
STATEMETHOD_END();

STATEMETHOD_HEADER( Complex_imag );
    STATEMETHOD_ACCESSNAMED( "Complex_imag", OBJ_NAMED_AT( BLOCK_SELF(), 1 ) );
STATEMETHOD_END();

STATEMETHOD_HEADER( Complex_imagK );
    STATEMETHOD_CHANGENAMED( "Complex_imagK", OBJ_NAMED_AT( BLOCK_SELF(), 1 ), MESS_ARG(0) );
STATEMETHOD_END();

BLOCKDEF_HEADER( Complex_asReal_block_1 )
    BLOCKDEF_VARS( "Complex_asReal_block_1", 0, 0, 3, BLOCK_IS_LITERAL );

    MESS_SEND( 0 );
        MESS_REC() = CNTXT_OFFSET( CN( CC ), 0 );   /*  CNTXT ( 1, 0)  */
        SEND_MESSAGE( 0, 79, -1 );   /* real */
    MESS_END( 0 );
    BLOCK_RESULT() = MESS_RESULT();
    BLOCKDEF_EXPLICIT_RETURN( BLOCK_IS_LITERAL );

BLOCKDEF_END();

BLOCKDEF_HEADER( Complex_asReal_block_2 )
    BLOCKDEF_VARS( "Complex_asReal_block_2", 0, 0, 3, BLOCK_IS_LITERAL );
    BLOCKDEF_EXPLICIT_RETURN( BLOCK_IS_LITERAL );

BLOCKDEF_END();

BLOCKDEF_HEADER( Complex_asReal )
    BLOCKDEF_VARS( "Complex_asReal", 0, 0, 3, BLOCK_IS_METHOD );

    MESS_SEND( 2 );
        ASSIGN_NEW_CLOSURE( MESS_ARG( 0 ), Complex_asReal_block_1, 0, CC );
        ASSIGN_NEW_CLOSURE( MESS_ARG( 1 ), Complex_asReal_block_2, 0, CC );
        MESS_SEND( 1 );
            MESS_ARG( 0 ) = LITERAL_AT_INDEX( 65 ); /* 0 */
            MESS_SEND( 0 );
                MESS_REC() = CNTXT_OFFSET( CC, 0 );  /*  CNTXT ( 0, 0)  */
                SEND_MESSAGE( 0, 81, -1 );  /* imag */
            MESS_END( 1 );
            MESS_REC() = MESS_RESULT();
            SEND_MESSAGE( 1, 19, -1 );  /* == */
        MESS_END( 1 );
        MESS_REC() = MESS_RESULT();
        SEND_MESSAGE( 2, -9, -1 );  /* ifTrue:ifFalse: */
    MESS_END( 1 );

    BLOCKDEF_DEFAULTRETURN( MESS_RESULT() );
BLOCKDEF_END();

BLOCKDEF_HEADER( Complex_class_realK_imagK )
    BLOCKDEF_VARS( "Complex_class_realK_imagK", 2, 0, 3, BLOCK_IS_METHOD );

    MESS_SEND( 1 );
        MESS_ARG( 0 ) = CNTXT_OFFSET( CC, 1 );  /*  CNTXT ( 0, 1)  */
        MESS_REC() = BLOCK_SELF().
        SEND_MESSAGE( 1, 80, -1 );  /* real: */
    MESS_END( 1 );
    MESS_SEND( 1 );
        MESS_ARG( 0 ) = CNTXT_OFFSET( CC, 2 );  /*  CNTXT ( 0, 2)  */
        MESS_REC() = BLOCK_SELF().
        SEND_MESSAGE( 1, 82, -1 );  /* imag: */
    MESS_END( 1 );

    BLOCKDEF_DEFAULTRETURN( MESS_RESULT() );
BLOCKDEF_END();

/***
 ***        NumbersModule_E1
 ***/

MODBLOCK_HEADER( NumbersModule_E1 )
    MODBLOCK_VARS( "NumbersModule_E1" );

    MESS_SEND( 2 );
        MESS_ARG( 0 ) = LITERAL_AT_INDEX( 65 ); /* 0 */
        MESS_ARG( 1 ) = LITERAL_AT_INDEX( 66 ); /* 1 */
        MESS_REC() = CNTXT_OFFSET( CC, 1 );  /*  CNTXT ( 0, 1)  */
        SEND_MESSAGE( 2, -1, 2 );  /* real:imag: */
    MESS_END( 1 );
    BLOCK_RESULT() = MESS_RESULT();
```

```
MODBLOCK_END();
/******
 ******      NumbersModule
 ******/

MODULE_DEF( NumbersModule )
    ASSIGN_MODULE_SPACE( 3, "NumbersModule" );

    ASSIGN_MODULE_IMPORT( 0, 1, 0 );
    ASSIGN_MODULE_CLASS( 1, ComplexClass );
    ASSIGN_MODULE_EXPRESSION( 2, NumbersModule_E1 );
MODULE_END();
```

The expanded C-code for the module is as follows:

```
int **
ComplexClass ()
{
   struct dict_t * inst[2];
   struct dict_t * class[2];
   struct set_t * superclasses;
   int ** result; ;

   result = (int **) makeClass( inst[0], inst[1], class[0], class[1], superclasses,
                        2, 0, 0, 0, 0, 0, 23 );
   return( result );;
};

int **
Complex_binary_K( self, args, size )
   int ** self;
   int *** args;
   int size;
{
   int ** result;
   int ** subBlockResult;
   int ** cntxt;

   if ( 1 == 1 )
   {
      {
         int ibc_i;
         {
            int ** mssz;
            {
               ( mssz ) = ( int **) myalloc( (2)+ 1+ 0+NBI , sizeof( int **) );
               ((int) ((  mssz )[1])) = ( ( (2)+ 1+ 0+NBI << 8) | ((2)+ 1+ 0 << 16) | ( (2)+ 1 << 24) );
            }
            (int **) ((int **) ( mssz )[0]) = (int **) (int **) (MS.hardcode[0 ]);
            ( (int **) (( mssz )[( (int) ( (2) ) )+ 0 ]) ) = (int **)   1 +  1 +  1 ;
            {
               (  cntxt) = ( int **) myalloc( (2)+ 2+1+NBI , sizeof( int **) );
               ((int) ((   cntxt)[1])) = ( ( (2)+ 2+1+NBI << 8) | ((2)+ 2+1 << 16) | ( (2)+ 2 << 24) );
            }
            (int **) ((int **) (  cntxt )[0]) = (int **) 0;
            ( (int **) ((  cntxt)[( (int) ( (2) ) )+ 0 ]) ) = (int **) 0;
            ( (int **) ((  cntxt)[( (int) ( (2) ) )+ 1 ]) ) = (int **) 0;
      {
        ( (int **) (  cntxt)[( (int) ( ( ( ((int) ((  cntxt)[1])) ) >> 24) ) )+0 ] ) =
         (int **) ( int ** *) myalloc( (int) (( (int **) ((  mssz )[( (int) ( (2) ) )+ 0 ]) )) + 1 , sizeof( int **) );
        ((int **)( ( (int **) (  cntxt)[( (int) ( ( ( ((int) ((  cntxt)[1])) ) >> 24) ) )+0 ] )))[0] = (int **) mssz ;
      }
    }
    ( ( (int **) ((  cntxt )[( (int) ( (2) ) )+ 0 ]) ) ) = ( (int **) ( MS.module[  3 ].context ) );
    ( ( (int **) ((  cntxt )[( (int) ( (2) ) )+ 1 ]) ) ) = (int **)  3;
    ((((int **)( ( (int **) ( cntxt)[( (int) ( ( ( ((int) (( cntxt)[1])) ) >> 24) ) )+0] )))[  0  +1]) ) = self;
    for ( ibc_i = 1; ibc_i <=  1; ibc_i++ )
    {
      ((((int **)( ( (int **) ( cntxt)[( (int) ( ( ( ((int) (( cntxt)[1])) ) >> 24) ) )+0] )))[  ibc_i  +1]) ) =
       (int **)(args[ibc_i-1]);
    }
    for ( ibc_i =  1 + 1; ibc_i < (1 +  1 +  1); ibc_i++ )
    {
      ((((int **)( ( (int **) ( cntxt)[( (int) ( ( ( ((int) (( cntxt)[1])) ) >> 24) ) )+0] )))[  ibc_i  +1]) ) =
       (int **) (MS.hardcode[10 ]);
    }
  }
}
else
{
  {
    int ibc_i;
    cntxt = MS.literalContext;
    for ( ibc_i = 0; ibc_i <  1; ibc_i++ )
    {
      ((((int **)( ( (int **) ( cntxt)[( (int) ( ( ( ((int) (( cntxt)[1])) ) >> 24) ) )+0] )))[  ibc_i  +1]) ) =
       (int **)(args[ibc_i]);
    }
    for ( ibc_i =  1; ibc_i < ( 1 +  1); ibc_i++ )
    {
      ( (((int **)( ( (int **) ( cntxt)[( (int) ( ( ( ((int) (( cntxt)[1])) ) >> 24) ) )+0] )))[  ibc_i  +1]) ) =
       (int **) (MS.hardcode[10 ]);
    }
  }
}
MS.context[MS.contextSize] = (int **)  cntxt ;
++(MS.contextSize);

{
  int ** rec;
  int *** args = ( int ** *) myalloc(  2  , sizeof( int **) );
  {
    int ** rec;
    int *** args = ( int ** *) myalloc(  1  , sizeof( int **) );
    {
      int ** rec;
      int *** args = ( int ** *) myalloc(  0  , sizeof( int **) );
      rec = (int **) ( (int **) (((int **)( ( (int **) ( ( MS.context[MS.contextSize-1] ))
          [( (int) ( ( ( ((int) ((( ( MS.context[MS.contextSize-1] ))[1])) ) >> 24) ) )+0] )))[  1  +1]) );
      subBlockResult = execute( rec, args,  0,  79,  -1  );
      myfree(  args  );
      if ( ( 1  != 1) && ( ( (int) ( ( ((int) (( subBlockResult )[1])) ) & 0x1 ) ) ) )
      {
        result = subBlockResult;
        {
          --(MS.contextSize);
```

```
            }
            return( result );
         }
      }
   args[  0  ] = subBlockResult;
   {
      int ** rec;
      int *** args = ( int ** *) myalloc(  0  , sizeof( int **) );
      rec = ((((int **)( ( (int **) ( ( MS.context[MS.contextSize-1] ))
                 [( (int) ( ( ( ((int) ((( ( MS.context[MS.contextSize-1] ))[1])) ) >> 24) ) )+0] )))[  0  +1]) );
      subBlockResult = execute( rec, args,  0,  79,  -1  );;
      myfree(  args  );
      if ( ( 1  != 1) && ( ( (int) ( ( ((int) (( subBlockResult )[1])) ) & 0x1 ) ) ) )
      {
         result = subBlockResult;
         {
            --(MS.contextSize);
         }
         return( result )
      }
   }
   rec = subBlockResult;
   subBlockResult = execute( rec, args,  1,  45,  -1  );
   myfree(  args  );
   if ( ( 1  != 1) && ( ( (int) ( ( ((int) (( subBlockResult )[1])) ) & 0x1 ) ) ) )
   {
      result = subBlockResult;
      {
         --(MS.contextSize);
      }
      return( result );
   }
}
args[  0  ] = subBlockResult;
{
   int ** rec;
   int *** args = ( int ** *) myalloc(  1  , sizeof( int **) );
   {
      int ** rec;
      int *** args = ( int ** *) myalloc(  0  , sizeof( int **) );
      rec = ((((int **)( ( (int **) ( ( MS.context[MS.contextSize-1] ))
                [( (int) ( ( ( ((int) ((( ( MS.context[MS.contextSize-1] ))[1])) ) >> 24) ) )+0] )))[  1  +1]) );
      subBlockResult = execute( rec, args,  0,  81,  -1  );
      myfree(  args  );
      if ( ( 1  != 1) && ( ( (int) ( ( ((int) (( subBlockResult )[1])) ) & 0x1 ) ) ) )
      {
         result = subBlockResult;
         {
            --(MS.contextSize);
         }
         return( result );
      }
   };
   args[  0  ] = subBlockResult;
   {
      int ** rec;
      int *** args = ( int ** *) myalloc(  0  , sizeof( int **) );
      rec = ((((int **)( ( (int **) ( ( MS.context[MS.contextSize-1] ))
           [( (int) ( ( ( ((int) ((( ( MS.context[MS.contextSize-1] ))[1])) ) >> 24) ) )+0] )))[  0  +1]) );
```

```
            subBlockResult = execute( rec, args, 0, 81, -1 );
            myfree( args );
            if ( ( 1 != 1) && ( ( (int) ( ( ((int) (( subBlockResult )[1])) ) & 0x1 ) ) ) )
            {
               result = subBlockResult;
               {
                  --(MS.contextSize);
               }
               return( result );
            }
         }
         rec = subBlockResult;
         subBlockResult = execute( rec, args, 1, 45, -1 );
         myfree( args );
         if ( ( 1 != 1) && ( ( (int) ( ( ((int) (( subBlockResult )[1])) ) & 0x1 ) ) ) )
         {
            result = subBlockResult;
            {
               --(MS.contextSize);
            }
            return( result );
         }
      }
      args[ 1 ] = subBlockResult;
      rec = ((((int **)( ( (int **) ( ( MS.context[MS.contextSize-1] ))
           [( (int) ( ( ( ((int) (( ( MS.context[MS.contextSize-1] ))[1])) ) >> 24) ) )+0] )))[ 1 +1]) );
      subBlockResult = execute( rec, args, 2, -1, 2 );
      myfree( args );
      if ( ( 1 != 1) && ( ( (int) ( ( ((int) (( subBlockResult )[1])) ) & 0x1 ) ) ) )
      {
         result = subBlockResult;
         {
            --(MS.contextSize);
         }
         return( result );
      }
   }
   ((((int **)( ( (int **) ( ( MS.context[MS.contextSize-1] ))
        [( (int) ( ( ( ((int) (( ( MS.context[MS.contextSize-1] ))[1])) ) >> 24) ) )+0] )))[ 2 +1]) ) =
          subBlockResult;
   result = ((((int **)( ( (int **) ( ( MS.context[MS.contextSize-1] ))
        [( (int) ( ( ( ((int) (( ( MS.context[MS.contextSize-1] ))[1])) ) >> 24) ) )+0] )))[ 2 +1]) );

   if ( 1 != 1 )
   {
      ((int) (( result )[1])) = ( (((int) (( result )[1]))) | 1 );
   }
   {
      --(MS.contextSize);
   }
   return( result );;
};


int ** Complex_real( self, args, size )
   int ** self;
   int *** args;
   int size;
```

```
{
   return( (int **) ( ( (int **) (( self)[( (int) ( (2) ) )+ 0 ]) ) ) );
}

int **  Complex_realK( self, args, size )
   int ** self;
   int *** args;
   int size;
{
   ( ( (int **) (( self)[( (int) ( (2) ) )+ 0 ]) )) = ( args[ 0 ] ); return( (int **) ( args[ 0 ] ) );
};

int **  Complex_imag( self, args, size )
   int ** self;
   int *** args;
   int size;
{
   return( (int **) ( ( (int **) (( self)[( (int) ( (2) ) )+ 1 ]) ) ) );
};

int **  Complex_imagK( self, args, size )
   int ** self;
   int *** args;
   int size;
{
   ( ( (int **) (( self)[( (int) ( (2) ) )+ 1 ]) )) = ( args[ 0 ] ); return( (int **) ( args[ 0 ] ) );
};

int **  Complex_asReal_block_1( self, args, size )
   int ** self;
   int *** args;
   int size;
{
   /* ... */
};

int **  Complex_asReal_block_2  ( self, args, size )
   int ** self;
   int *** args;
   int size;
{
   /* ... */
};

int **  Complex_asReal  ( self, args, size )
   int ** self;
   int *** args;
   int size;
{
   /* ... */
};

int **  Complex_class_realK_imagK  ( self, args, size )
   int ** self;
   int *** args;
   int size;
{
   /* ... */
};
```

```
int **
NumbersModule_E1()
{
   int ** result;
   int ** subBlockResult;

   {
    int ** rec;
    int *** args = ( int ** *) myalloc(  2  , sizeof( int **) );
    args[  0  ] = MS.literals[ 65 ];
    args[  1  ] = MS.literals[ 66 ];
    rec = ((((int **)( ( (int **) ( ( MS.context[MS.contextSize-1] ))
             [( (int) ( ( ( ((int) (( ( MS.context[MS.contextSize-1] ))[1])) ) >> 24) ) )+0] )))[  1  +1]) );
    subBlockResult = execute( rec, args,  2,  -1,  2  );
    myfree(  args  );
    if ( ( 1  != 1) && ( ( (int) ( ( ((int) (( subBlockResult )[1])) ) & 0x1 ) ) ) )
    {
       result = subBlockResult;
       {
        --(MS.contextSize);
       }
       return( result );
    }
   };
   result = subBlockResult;
   return( result ); };
}

void
NumbersModule( result )
   struct msmodule_t * result;
{
   {
      int ** cntxt;
      {
         int ** mssz;
         {
            ( mssz ) = ( int **) myalloc( (2)+ 1+ 0+NBI , sizeof( int **) );
            ((int) ((  mssz )[1])) = ( ( (2)+ 1+ 0+NBI << 8) | ((2)+ 1+ 0 << 16) | ( (2)+ 1 << 24) );
         }
         (int **) ((int **) ( mssz )[0]) = (int **) (int **) (MS.hardcode[0 ]);
         ( (int **) (( mssz )[( (int) ( (2) ) )+ 0 ]) ) = (int **)    3 ;
         {
            (  cntxt) = ( int **) myalloc( (2)+ 2+1+NBI , sizeof( int **) );
            ((int) ((   cntxt)[1])) = ( ( (2)+ 2+1+NBI << 8) | ((2)+ 2+1 << 16) | ( (2)+ 2 << 24) );
         }
         (int **) ((int **) (  cntxt )[0]) = (int **) 0;
         ( (int **) ((  cntxt)[( (int) ( (2) ) )+ 0 ]) ) = (int **) 0;
         ( (int **) ((  cntxt)[( (int) ( (2) ) )+ 1 ]) ) = (int **) 0;
         {
            ( (int **) (  cntxt)[( (int) ( ( ( ((int) ((  cntxt)[1])) ) >> 24) ) )+0 ] ) =
             ( int ** *) myalloc( (int) (( (int **) ((  mssz )[( (int) ( (2) ) )+ 0 ]) )) + 1 , sizeof( int **) );
            ((int **)( ( (int **) (  cntxt)[( (int) ( ( ( ((int) ((  cntxt)[1])) ) >> 24) ) )+0 ] )))[0] = (int **) mssz ;
         }
      }
      result->context = cntxt;
      MS.contextSize = 0;
      if ( MS.contextSize == 0 )
      {
        MS.context[0] =  result->context ;
        MS.contextSize = 1;
      }
    }
    {
      ((((int **)( ( (int **) ( result->context)[( (int) ( ( ( ((int) (( result->context)[1])) ) >> 24) ) )+0] )))[   0  +1]) ) =
       ((((int **)( ( (int **) ( MS.module[ 1].context)
                   [( (int) ( ( ( ((int) (( MS.module[ 1].context)[1])) ) >> 24) ) )+0] )))[   0   +1]) );
    }
    {
      ((((int **)( ( (int **) ( result->context)[( (int) ( ( ( ((int) (( result->context)[1])) ) >> 24) ) )+0] )))[   1  +1]) ) =
       (int **)  ComplexClass  ();
    }
    {
      ((((int **)( ( (int **) ( result->context)[( (int) ( ( ( ((int) (( result->context)[1])) ) >> 24) ) )+0] )))[   2  +1]) ) =
       (int **)  NumbersModule_E1  ( );
    }
};
```

Appendix D : Object Module and Kernel Module Source Code

The *Object* module was obtained by dividing the behavior of the Smalltalk Object class into related areas. Since methods within Object depend on the existence of other classes, those classes must be defined and placed in appropriate modules to be imported by the Object module. In order to avoid having to implement the entire Smalltalk system library, only methods that use fundamentally important classes are implemented. Fundamentally important classes consists of classes to perform I/O (something similiar to the Stream hierarchy in ST80) and basic behavior in collection classes.

```
{
    module 'Object'


        MemoryObject (public) ->
        {
                class { refines nil }

                instance
                { behavior
                        release -> primitive
                }
        }

        TestableObject (public) ->
        {
                class { refines nil }

                instance
                { behavior
                        isInteger -> primitive
                        isNil -> primitive
                        isSequenceable -> primitive
                        isString -> primitive
                        isSymbol -> primitive
                        isLiteral -> primitive
                        notNil -> primitive
                        respondsToArithmetic -> primitive
                }
        }

        ComparableObject (public) ->
        {
                class { refines nil }

                instance
                { behavior
                        = -> primitive
                        == -> primitive
                        ~= -> primitive
                        ~~ -> primitive
        }
}

CopyableObject (public) ->
{
        class { refines nil }

        instance
        { behavior
                deepCopy -> primitive
                shallowCopy -> primitive
                copy -> method [ ^self shallowCopy postCopy ]
                postCopy -> [ ^self ]
        }
}

ClassableObject (public) ->
{
        class { refines nil }

        instance
        { behavior
                class -> primitive
                species -> method
                [
                        "Answer the preferred class for reconstructing the receiver.
                         For example, collections create new collections whenever
                         enumeration messages such as collect: or select: are invoked.
                         The new kind of collection is determined by the species of the
                         original collection.  Species and class are not always the
                         same.  For example, the species of Interval is Array."

                        ^self class
                ]

                isKindOf: -> method
                [ :aClass |
                        "Answer a Boolean as to whether the class, aClass, is a
                         superclass or class of the receiver."

                        self class == aClass
                                ifTrue: [^true]
                                ifFalse: [^self class inheritsFrom: aClass]
                ]

                isMemberOf: -> method [ :aClass | ^self class == aClass ]

                respondsTo: -> method
                [ :aSymbol |
                        "Answer a Boolean as to whether the method dictionary of the
                         receiver's class contains aSymbol as a message selector."

                        ^self class canUnderstand: aSymbol
                ]
        }
}

CreatableObject (public) ->
{
        class { refines nil }

        class
        { behavior
                new -> primitive
        }
}

PerformableObject (public) ->
{
        class { refines nil }

        instance
        { behavior
                perform: -> primitive
                perform:with: -> primitive
                perform:with:with: -> primitive
                perform:with:with:with: -> primitive
                perform:withArguments: -> primitive
        }
}

ErrorHandlingObject (public) ->
{
        class { refines nil }

        instance
        { behavior
                doesNotUnderstand: -> primitive
                error: -> method
                [ :aString |
                ]

                errorSignal -> method []
                [
                        " Answer the Signal used for miscellaneous errors "
                        " Currently just returns nil"
                ]

                halt -> primitive
                "Call the ST80 halt method"
        }
}

PrintableObject (public) ->
{
        class { refines nil }

        instance
        { behavior
                printString -> primitive

                printOn: -> method
                [ :aStream |
                        "Append to the argument aStream a sequence of characters that
                         identifies the receiver."

                        | title |
            title := self class name.
            aStream nextPutAll: ((title at: 1) isVowel
            ifTrue: ['an ']
            ifFalse: ['a ']) , title
        ]

        storeOn: -> method [ ]
        [ :aStream |
            "Append to the argument aStream a sequence of characters that is an
             expression whose evaluation creates an object similar to the receiver.
            "
            aStream nextPut: $(.
            self class isVariable
            ifTrue:
            [
                aStream nextPutAll: '(', self class name, ' basicNew: ';
                store: self basicSize;
                nextPutAll: ') '
            ]
            ifFalse:
           [
                aStream nextPutAll: self class name, ' basicNew'
            ].

            1 to: self class instSize do:
            [:i |
                aStream nextPutAll: ' instVarAt: ';
                store: i;
                nextPutAll: ' put: ';
                store: (self instVarAt: i);
                nextPut: $;
            ].

            1 to: self basicSize do:
            [:i |
                aStream nextPutAll: ' basicAt: ';
                store: i;
                nextPutAll: ' put: ';
                store: (self basicAt: i);
                nextPut: $;
            ].
            aStream nextPutAll: ' yourself)'
        ]

        storeString -> method []
        [
            "Answer a String representation of the receiver from which the
             receiver can be reconstructed."

            | aStream |
            aStream := WriteStream on: (String new: 16).
            self storeOn: aStream.
            ^aStream contents
        ]
    }
}

IOObject (public) ->
{
```

```
            class { refines nil }

            instance
            { behavior
                    outputString: -> primitive
            }
    }

    Object (public) ->
    {
            class
            { refines
                    MemoryObject
                    TestableObject
                    ComparableObject
                    CopyableObject
                    ClassableObject
                    CreatableObject
                    PerformableObject
                    ErrorHandlingObject
                    PrintableObject
                    IOObject
            }
    }
}
```

The Kernel module defines all required classes and required methods, as well as adding methods required due to the implementation strategy choosen. There is some question as to whether these required classes should inherit from Object or some other class (like CreateableObject). On one hand, it is better to allow the programmer to determine exactly what behavior is to be inherited, but on the other hand, libraries exist to reduce the amount of work programmers have to do. Because of the class extension facility, the final MS library structure will probably define a base kernel module which does not inherit from anything (except maybe CreateableObject), and then define a ExtendedKernel module which refines and adds superclasses.

The behavior defined in this module is expected to be expanded substantial as it is more fully developed. Also, superclasses besides Object will be defined in other modules and imported here. Such modules will by definition be part of the Kernel Group.

```
{
        module 'Kernel'

        Object -> { from 'Object' }
```

```
Integer (public) ->
{
        class { refines Object  }

        instance
        {  behavior
                + -> primitive
                - -> primitive
                * -> primitive
                // -> primitive
                \ -> primitive
                = -> primitive
                ~= -> primitive
                < -> primitive
                > -> primitive
                <= -> primitive
                >= -> primitive
                to:do: -> primitive
        }

}

Array (public) ->
{
        class { refines Object }

        instance
        {  behavior
                { size size: | at: at:put: } -> variable
        }
}

Character (public) ->
{
        class { refines Object }

        instance
        { behavior
                value: -> primitive
        }
}

Closure (public) ->
{
        class { refines Object }

        instance
        {  behavior

                "Order of specification of state doesn't matter - hardcoded.
                 user can change these around without problems.  However,
                 what if use wants to add behavior?  Must somehow know what the
                 starting index for variables is!!! "

                { block block: } -> variable
                { context context: } -> variable

                whileTrue: ->
                [
                ^(self value)
                        ifTrue:
                        [
                                aBlock value.
                                [self value]
                                        whileTrue:
                                        [
                                                aBlock value
                                        ].
                        ].
                ]

                value -> primitive
                value: -> primitive
                valueWithArgs: -> primitive

                messageNotUnderstood: -> primitive
                invalidArgumentCount: -> primitive
        }
}

Float (public) ->
{
        class { refines Object }
}

Message (public) ->
{
        class { refines Object }

        instance
        { behavior
                { selector selector: } -> variable
                { args args: } -> variable
        }
}

MethodSelector (public) ->
{
        class { refines Object }

        instance
        {  behavior
                { name name: } -> variable
        }
}

String (public) ->
{
        class { refines Object }

        instance
        {  behavior
                { size size: | at: at:put: } -> binary
        }
}

UndefinedObject (public) ->
{
```

```
                class { refines Object }
        }

        True (public) ->
        {
                class { refines Object }

                instance
                {  behavior
                        ifTrue: -> primitive
                        ifFalse: -> primitive
                }
        }

        False (public) ->
        {
                class { refines Object }

                instance
                {  behavior
                        ifTrue: -> primitive
                        ifFalse: -> primitive
                }
        }
}
```

Appendix E : Implementation Dependent Issues

1. Immutable Literals

Literals are immutable, but the result of attempting to modify them is implementation dependent.

To handle the immutability of integers, characters and floats, their representation within a MS program is stateless — no state methods are declared for them). Obviously the underlying implementation must provide space to save different instances, but this space is not accessible from within MS.

However, Strings, Arrays and Methodselectors do have state. Literal instances of such classes are marked as immutable. The changing methods for these classes are primitives which check to insure that the object is not immutable; if it is, an MS run-time error is generated. The exact form of this error has not been determined.

2. Unrecognized Messages

The language specifications states if a message is not understood, the receiver of the message send is to be sent the message *doesNotUnderstand:withArguments:*, except when the message that is not understood is *doesNotUnderstand:withArguments:*, in which case the results are implementation independent. A generic MS 'implementation-dependent' runtime error is generated.

3. Incorrect Argument Counts

If a method is executed, but the number of actual arguments does not match the number of formal parameters, the results are implementation dependent.

The action performed is similiar to that for unrecognized messages. The message *invalidArgumentCount:* will be sent to the receiver. If this method is not understood, a generic MS 'incorrect argument count' runtime error will be generated.

4. Extended Character Classes

The language specification allows implementations to extend the character classes.

The current implementation provides only those classes specified.

5. Nested Assignment Order

The language specification allows assignment statements to be nested, so that a single expression is assigned to one or more variables. The order in which the assignment to variables takes place is implementation dependent.

The current implementation assigns values from right to left. However, since the order in which values is assign is completely irrelevant, any ordering is possible.

6. Indexed Variable Size Changing

When specifying the size of an indexed instance variable, if the argument to the size changing method is anything other than a non-negative integer, the results are implementation-dependent.

A runtime error is generated. This error will not provide the ability to continue runtime execution until the error is corrected.

7. Indexed Variable Size Accessing

When accessing the size of an indexed instance variable, the return value is 0 if the associated size changing message as not yet been called, and is otherwise equal to the integer argument to the size changing method. However, if the argument to the size changing method was not an valid integer, the return value is implementation dependent.

Note that even if an error is generated within the size changing method for illegal size arguments, it is still possible for the above situation to occur if the run-time environment error allows one to continue (although why it would is questionable). This error will never occur in current implementation.

8. Accessing Index of Indexable Variable

When accessing the index of an indexable variable, the results are implementation dependent if the index argument is not a valid integer or if the size argument to the last call of the associated size changing method was not a valid integer.

A runtime error is generated if a non-valid integer argument is given. The second possibility will never occur, due to the handling of non-valid integers to size changing state methods.

Note that the language specification should be more rigorous in its description of a valid integer. A valid integer is one between 0 and 'size-1' inclusive.

9. Changing Index of Indexable Variable

When changing the value of an index of an indexable variable, the results are implementation dependent if the index is not valid, the size is not a valid integer, or if the variable is binary and the value to be stored into the index is not an integer between 0 and 255 inclusive.

A non-continuable runtime is generated for cases 1 and 3. Case 2 will never occur.